Security Audit Report

Nawa Finance

v1.0

October 15, 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 Nawa Labs Ltd to perform a security audit of Smart Contract Audit of Nawa Finance.

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/Nawa-Finance/stzig-vault
Commit	`1da8194aa62aac8a6ea35d15cc4697b534b3efcd`
Scope	The scope is restricted to the following directories: • `contracts/nft` • `contracts/vault` • `packages/cw721`
Fixes verified at commit	`b2138ce6e5a3107554f6842f9b6466db4f63f1c9`

	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 stZIG Vault is a CosmWasm-based DeFi system for staking ZIG tokens, where users deposit uZIG and receive CW721-compliant NFTs representing their stZIG positions.

Each NFT encodes staking details and must be burned to initiate withdrawals, which then follow an unbonding process with slashing adjustments and configurable fees applied.

The vault integrates with external staker contracts for staking, tracks historical slashing to ensure fair withdrawals, and operates on 22-day automated cycles that handle staking, unbonding, and slashing 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	-
Code readability and clarity	Medium-High	Code structure is clear, modular, and aligned with CosmWasm conventions.
Level of documentation	Medium-High	Documentation is available and sufficient for understanding, though not exhaustive.
Test coverage	Medium	Test coverage reported by `cargo tarpaulin` is `66.36%`.

Summary of Findings

No	Description	Severity	Status
1	Missing caller authentication in `execute_receive_nft` allows malicious actors to steal funds from legitimate depositors	Critical	Resolved
2	Unbounded iteration on withdrawals leads to DoS	Critical	Resolved
3	Outdated slashing factors enable incorrect withdrawal computations	Critical	Resolved
4	Untrusted `user_address` in `execute_receive_nft` enables unauthorized withdrawals	Critical	Resolved
5	Incompatible deserialization of `FundsRaisedResponse` blocks slash factor updates	Critical	Resolved
6	Missing reservation mechanism in `execute_process_withdrawal` enables over-commitment and inconsistent payouts	Critical	Resolved
7	Incorrect loss percentage calculation	Critical	Resolved
8	Inconsistent scaling and producing incorrect slashing factor	Critical	Resolved
9	Unbounded withdrawal scanning enables DoS	Critical	Resolved
10	Missing price bound checks in `Deposit` and `ReceiveNft` enable sandwich attacks	Major	Resolved
11	Unfair withdrawal operation in protocol shortfall	Major	Resolved
12	Mutable NFT contract registration leads to an inconsistent state	Minor	Resolved
13	Missing input validation allows invalid fee percentage	Minor	Resolved
14	Centralized control allowing indefinite contract pausing	Minor	Acknowledged
15	Unbounded ledger summation risks	Minor	Resolved
16	Inefficient withdrawal processing loop enabling gas waste and potential DoS	Minor	Resolved

17	Unpaginated withdrawal requests query risks excessive gas use	Minor	Resolved
18	Redundant treasury entry points leading to code duplication	Informational	Resolved
19	Redundant pause logic leading to code duplication	Informational	Resolved
20	Contracts should implement a two-step ownership transfer	Informational	Resolved
21	Misleading error in NFT burn execution	Informational	Resolved
22	Missing enforcement of treasury address during instantiation	Informational	Resolved
23	Missing NFT contracts interface validation	Informational	Resolved
24	Variable marked as unused but actually used in withdrawal processing	Informational	Resolved
25	Missing balance validation before sending `stZIG` tokens in redeem operation	Informational	Resolved

Detailed Findings

1. Missing caller authentication in `execute_receive_nft` allows malicious actors to steal funds from legitimate depositors

Severity: Critical

In `contracts/vault/src/contract/exec.rs:262`, the `execute_receive_nft` entry point processes a withdrawal based solely on a user-supplied message and does not authenticate the caller.

Because there is no check that `info.sender` equals the registered `NFT_CONTRACT`, any external account or contract can invoke `execute_receive_nft` and pass an arbitrary `user_address` and `token_id`.

A malicious actor can therefore trigger withdrawals on behalf of victims and steal their funds.

Additionally, deposits are not removed or marked as consumed after a redemption. This allows a malicious or compromised NFT contract to repeatedly invoke `execute_receive_nft` with the same `token_id`, draining funds multiple times from the same deposit.

Recommendation

We recommend enforcing strict caller authentication and binding the redemption to a trusted state:

- Verify `info.sender` is equal to `NFT_CONTRACT` before any parsing or state changes, reject otherwise.
- Track deposits with unique identifiers and mark them as redeemed upon use. Reject any attempt to reuse the same identifier.

Status: Resolved

2. Unbounded iteration on withdrawals leads to DoS

Severity: Critical

In `contracts/vault/src/contract/exec.rs:389-442`, the `execute_process_unbonded` queries Valdora for all completed unbondings and then constructs and sends a `ValdoraExecMsg::WithdrawUnbonded` message to process them.

Per the `README`, the vault has an unlimited withdrawal request allowance with Valdora.

As a result, the number of iterations performed inside `WithdrawUnbonded` is unbounded.

Consequently, the protocol would accumulate over time a large number of withdrawal requests that would cause the execution to run out of gas, permanently disabling withdrawals.

Additionally, this can be quickly forced by a malicious actor who can mint many low-value NFTs and unstake them to create a large backlog of tiny unbondings. When processed, the single unbounded withdrawal call can exceed the block gas limit, consistently reverting and effectively bricking the redemption path.

This issue is also present in `packages/ledger/src/contract/exec.rs:467`. While out of scope, it should be considered when resolving the issue mentioned above.

Recommendation

We recommend introducing a batched flow that processes at most `limit` unbondings per call, persisting a cursor to resume progress safely across transactions.

Status: Resolved

3. Outdated slashing factors enable incorrect withdrawal computations

Severity: Critical

The contract processes slashes through a dedicated `ApplySlashes` entry point, separate from other operations such as `ReceiveNft`.

This separation means that `SLASH_FACTORS` and `EPOCH` values are not guaranteed to be updated when user-facing actions are executed. As a result, withdrawals may be calculated using stale or invalid slashing data, allowing incorrect redemption amounts to be constructed.

An attacker could exploit this by triggering withdrawals before slashing is applied, extracting more value than intended, and undermining protocol accounting integrity.

Recommendation

We recommend tightly coupling slashing updates with operations that depend on them by invoking or enforcing an update of `SLASH_FACTORS` whenever they are used in sensitive flows.

Status: Resolved

4. Untrusted `user_address` in `execute_receive_nft` enables unauthorized withdrawals

Severity: Critical

In `contracts/vault/src/contract/exec.rs:262`, the `execute_receive_nft` deserializes `user_address` from an arbitrary payload and uses it to determine the beneficiary.

Because `user_address` is entirely user-controlled, a malicious actor can craft a message that redeems to an attacker-controlled address, resulting in fund diversion and arbitrary asset withdrawal.

The function already receives a trusted context (`_sender`) from the CW721 hook but ignores it, amplifying the risk.

Recommendation

We recommend using `_sender` as the authoritative user identifier and not accepting `user_address` from the message payload.

Status: Resolved

5. Incompatible deserialization of `FundsRaisedResponse` blocks slash factor updates

Severity: Critical

In `contracts/vault/src/contract/exec.rs:503`, the `execute_apply_slashes` function queries Valdora's `FundsRaised` endpoint and attempts to deserialize the response into `FundsRaisedResponse`, which expects a field `amount` of type `Uint256`.

However, the Valdora contract actually returns `FundsRaisedResponse`, which uses `funds_raised` as the JSON key. Since Serde treats `amount` as a required field and it is absent, deserialization fails.

As a result, slash factor updates cannot be processed, leaving the protocol in a state where penalties are not applied.

Recommendation

We recommend aligning the response schema with the Valdora contract to prevent deserialization failures.

Status: Resolved

6. Missing reservation mechanism in `execute_process_withdrawal` enables over-commitment and inconsistent payouts

Severity: Critical

In `contracts/vault/src/contract/exec.rs:541-601`, the implementation of `execute_process_withdrawal` derives the available balance directly from the contract's live bank balance and only decrements a local variable when marking withdrawal requests as completed.

Because no reservation state is stored, the function can be executed multiple times, committing more withdrawals than the contract's true balance allows.

Consequently, users would not be able to claim their funds and malicious actors could exploit this to de-prioritize legit requests.

Furthermore, since no reserved amounts are persisted, the payout at `execute_claim` is recalculated using the then-current `SLASH_FACTORS`. This creates the possibility that a user's final claim differs from the originally intended value if slash factors change between processing and payout.

Together, these flaws enable economic inconsistencies, over-commitment of funds, and potential denial of service scenarios.

Recommendation

We recommend implementing a reservation mechanism to guarantee that withdrawals are committed safely and deterministically.

This can be achieved by introducing a `TOTAL_RESERVED` counter to track all committed withdrawal amounts and by persisting a per-request `reserved_amount`. At processing time, the available balance should be computed as the on-chain bank balance minus `TOTAL_RESERVED`. When a withdrawal is processed, the request should be marked as completed, its reserved amount recorded, and the `TOTAL_RESERVED` counter incremented accordingly.

At claim time, the payout should be made based on the stored reserved amount rather than recalculating it, with a fallback to recomputation only for legacy requests without a reserved value. This approach prevents repeated over-commitment across multiple executions and ensures that user claims remain stable regardless of later changes to slash factors.

Status: Resolved

7. Incorrect loss percentage calculation

Severity: Critical

In `contracts/vault/src/contract/exec.rs:507-519`, the vault computes the loss percentage by dividing the total slashes by the funds raised using integer `Uint256` division.

Because the total slashes value is the raw sum of per-ledger `slash_today` amounts and is not pre-scaled, the result is truncated toward zero whenever the slashes are smaller than the total funds raised.

In practice, this means the calculated percentage will almost always return zero, preventing the system from applying slashing correctly.

By contrast, the Valdora contract defines this percentage in basis points, multiplying the loss by `10,000` before dividing by the total, which preserves precision. The vault's failure to follow this model leads to inaccurate slash factor updates and undermines the intended penalty mechanism.

Recommendation

We recommend updating the calculation to follow the same basis point approach as the Valdora contract. Additionally, we recommend implementing strict integration tests.

Status: Resolved

8. Inconsistent scaling and producing incorrect slashing factor

Severity: Critical

In `contracts/vault/src/contract/exec.rs:521-532`, the computation of `today_factor` uses a `RAY` scale of one billion and a formula that diverges from Valdora's reference implementation.

Since Valdora defines slashing math in `RAY` with eighteen decimals and applies a specific update formula, any deviation in scale or algebra here will yield inaccurate `SLASH_FACTORS`.

This misalignment can propagate across epochs, corrupt staking economics, and cause inconsistent redemption amounts.

Recommendation

We recommend strictly matching Valdora's slashing specification by adopting a shared `RAY` constant with eighteen decimals and implementing the exact formula used by Valdora.

Status: Resolved

9. Unbounded withdrawal scanning enables DoS

Severity: Critical

In `contracts/vault/src/contract/exec.rs:541-609`, the `execute_process_withdrawal` loads, filters, and sorts the entire `WITHDRAWAL_REQUESTS` collection on each call, then iterates over all entries, including already completed ones, to mark additional requests as completed based on the contract's available ZIG balance.

As the set grows, this unbounded, repeated full-scan pattern drives gas costs upward and eventually exceeds block limits.

An attacker can accelerate this by submitting many small requests, causing the function to consistently run out of gas and rendering withdrawal processing permanently unusable.

Recommendation

We recommend having two maps, one for pending and one for completed requests. Also, we recommend implementing batching.

Status: Resolved

10. Missing price bound checks in `Deposit` and `ReceiveNft` enable sandwich attacks

Severity: Major

The `Deposit` and `ReceiveNft` execution paths directly rely on prices retrieved from Valdora without enforcing any slippage tolerance or target price constraints.

This design leaves both operations vulnerable to sandwich attacks, where a malicious actor can manipulate the price around the user's transaction.

Recommendation

We recommend allowing users to specify a minimum received amount for deposits or a maximum acceptable price impact for withdrawals.

Status: Resolved

11. Unfair withdrawal operation in protocol shortfall

Severity: Major

In the `execute_process_withdrawal` function in `contracts/vault/src/contract/exec.rs:580-588`, the withdrawal processing

logic implements an algorithm that processes requests in chronological order until the vault's available balance is exhausted. When the vault runs out of funds, the loop terminates immediately, leaving all remaining withdrawal requests unprocessed.

While this is an unlikely condition where the vault cannot meet the demand of the requested withdrawals, it likely means that the protocol is insolvent and has occurred some greater loss that requires a manual intervention. So in this case the `execute_process_withdrawal` should error and pause until the loss has been reconciled, or the loss can be socialized across all users rather than giving early withdrawers a full amount and effectively giving the latter ones nothing.

If this situation were to occur, the loop would skip over larger amounts that exceed the balance and will continue to redeem any smaller amounts possible.

Additionally, this issue is also present in `packages/staker/src/contract/exec.rs:1681`. While this is out of scope, it should be considered when resolving the issue mentioned above.

Recommendation

We recommend running the loop to process the withdrawals and if the amount left is less than the required amount then to trigger an error and pause the protocol to allow for the balances to be properly reconciled.

Status: Resolved

12. Mutable NFT contract registration leads to an inconsistent state

Severity: Minor

In `contracts/vault/src/contract/exec.rs:84-110`, the `execute_register_nft` function allows the administrator to register or replace the `NFT_CONTRACT` at any time during runtime.

Since the NFT contract reference defines core protocol behavior and is linked to the data stored in the contract, its mutability significantly undermines system integrity.

Recommendation

We recommend removing the `execute_register_nft` function entirely and enforcing immutability of the `NFT_CONTRACT` after initialization. The NFT contract address should be set only once during contract instantiation and must remain unchanged throughout the contract's lifecycle. If a change is ever required, it should only occur through a controlled migration process, ensuring the correctness of the data stored in the contract.

Status: Resolved

13. Missing input validation allows invalid fee percentage

Severity: Minor

In `contracts/vault/src/contract/init.rs:32-34`, the contract initializes `fee_percentage` using `msg.fee_percentage.unwrap_or(1000u64)` without enforcing boundary conditions.

Since no validation is performed, it would be possible to set `fee_percentage` to zero (disabling fees entirely) or to a value greater than `10,000`, leading to miscalculated fees, unexpected contract states, or denial of service (DoS) scenarios where transactions are rejected due to invalid fee computations.

Recommendation

We recommend enforcing strict input validation by ensuring that `fee_percentage` values are greater than zero and strictly less than `10,000`.

Status: Resolved

14. Centralized control allowing indefinite contract pausing

Severity: Minor

In `contracts/vault/src/contract/exec.rs:10-58`, the `execute_pause` and `execute_unpause` functions grant the administrator unilateral control over the contract's operational state.

This centralization allows the admin to pause the contract indefinitely, effectively halting all functionality without any time limits or external checks.

Such unrestricted control introduces a single point of failure and could be abused, either maliciously or unintentionally, leading to a prolonged denial of service (DoS) for all users.

Recommendation

We recommend introducing safeguards against indefinite pausing. Possible mitigations include:

- Enforcing a maximum pause duration, after which the contract automatically resumes normal operation.
- Implementing multi-signature approval or governance mechanisms for pause and unpause actions.

Status: Acknowledged

15. Unbounded ledger summation risks

Severity: Minor

In `contracts/vault/src/contract/exec.rs:446-470`, the `get_slash` function queries Valdora for all registered ledgers and then iterates over the entire collection to compute the sum of `slash_today` values.

Since the number of ledgers in Valdora is unbounded, the operation may exceed block gas limits as the system scales.

This creates a potential denial of service (DoS) vector where the function becomes unusable, preventing the protocol from updating or applying slash factors.

Recommendation

We recommend implementing tests to benchmark this execution path and monitor the number of registered ledgers on Valdora.

Status: Resolved

16. Inefficient withdrawal processing loop enabling gas waste and potential DoS

Severity: Minor

In `contracts/vault/src/contract/exec.rs:580-602`, the `execute_process_withdrawal` loop scans the entire `sorted_requests` vector on every invocation, even when the remaining `available_balance` is too small to satisfy any request.

Given the protocol's deposit minimum, scenarios with a very low available balance trigger full-vector iterations that can never complete a match.

This results in unnecessary storage reads and computation, inflating gas consumption and creating a denial of service (DoS) vector when an attacker accumulates many pending requests that all exceed the residual balance.

Recommendation

We recommend checking a configured minimum payable amount and returning immediately if the available balance is below that threshold.

Status: Resolved

17. Unpaginated withdrawal requests query risks excessive gas use

Severity: Minor

In `contracts/vault/src/contract/exec.rs:580-602`, The `query_get_all_withdrawal_requests` function loads the full `WITHDRAWAL_REQUESTS` set and filters it in-memory by requester address.

This can return a very large payload and perform unbounded iteration, increasing gas and response size, and degrading node performance.

Although this is a read-only query, the lack of pagination creates avoidable resource pressure and hampers client UX.

Recommendation

We recommend implementing pagination.

Status: Resolved

18. Redundant treasury entry points leading to code duplication

Severity: Informational

In `contracts/vault/src/contract/exec.rs:111-168`, the contract exposes two entry points, specifically, `execute_set_treasury` and `execute_update_treasury`, that perform the same access control, validation, and state update on `TREASURY`.

Maintaining duplicate code paths for identical state transitions increases the risk of logic drift, overlooked checks, or inconsistent event attributes during future changes.

Additionally, these functions may cause issues with indexers and transaction history because the `execute_update_treasury` function does not actually enforce that it is updating the treasury, it could be called to actually set the treasury.

Recommendation

We recommend consolidating both functions into a single entry point.

Status: Resolved

19. Redundant pause logic leading to code duplication

Severity: Informational

In `contracts/vault/src/contract/exec.rs:10-58`, the contract defines two separate functions, `execute_pause` and `execute_unpause`, that implement nearly identical logic for toggling the paused state.

This duplication increases code complexity and maintenance overhead, while also raising the risk of inconsistent logic between the two functions.

A single unified function with an argument to determine the desired state (pause or unpause) would improve maintainability and reduce potential errors.

Recommendation

We recommend refactoring the contract to consolidate `execute_pause` and `execute_unpause` into a single function

Status: Resolved

20. Contracts should implement a two-step ownership transfer

Severity: Informational

The 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.

Recommendation

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

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

Status: Resolved

21. Misleading error in NFT burn execution

Severity: Informational

In `contracts/vault/src/contract/exec.rs:319-324`, within the NFT burn logic, failures when serializing the `burn_exec_msg` are incorrectly mapped to `NawaNFTError::NftMintFailed`.

This creates misleading error messages that obscure the actual root cause, making it difficult for developers, operators, and auditors to distinguish between minting failures and serialization/burn execution errors.

Recommendation

We recommend introducing a dedicated error variant to accurately represent failures in this context.

Status: Resolved

22. Missing enforcement of treasury address during instantiation

Severity: Informational

The vault designates the treasury address as an optional parameter at instantiation, but in practice, the contract logic requires it to be present.

For example, claims fail if the treasury is unset, since the code attempts to load the treasury address and raises an error when it is missing.

This creates an inconsistency between the contract specification and its actual behavior, leaving the system in a non-functional state if no treasury is provided at deployment.

Recommendation

We recommend enforcing the presence of a valid treasury address at instantiation.

Status: Resolved

23. Missing NFT contracts interface validation

Severity: Informational

In `contracts/vault/src/contract/exec.rs:99`, the function does not validate that the specified contract meets the required interface.

This could mean that the caller could accidentally pass any address as the parameter and this error would not be known until the vault dispatched messages to the incorrect contract.

Recommendation

We recommend implementing a query that can have its results validated to ensure that the specified contract implements the expected interface.

Status: Resolved

24. Variable marked as unused but actually used in withdrawal processing

Severity: Informational

In the `execute_receive_nft` function, defined in `contracts/vault/src/contract/exec.rs:305`, the variable `_deposit_info` is prefixed with an underscore, indicating it is intentionally unused.

However, it is actually used in line `330` to extract the `stzig_amount` for the redeem operation. This creates confusion about the variable's purpose and violates Rust naming conventions.

Recommendation

We recommend removing the underscore prefix from `_deposit_info` in line `305` to properly indicate that the variable is being used in the function.

Status: Resolved

25. Missing balance validation before sending `stZIG` tokens in redeem operation

Severity: Informational

In the `execute_receive_nft` function, defined in `contracts/vault/src/contract/exec.rs:330-340`, the code attempts to send `stzig_amount` tokens with the redeem message without first validating that the contract has sufficient `stZIG` balance.

While the contract should theoretically have the required balance (since it's burning an NFT that represents `stZIG` ownership), there's no explicit check to ensure the balance exists before attempting the transfer. If the contract somehow lacks the required `stZIG` balance, the transaction would fail with a generic error that would be difficult to debug, as the failure would occur in the Valdora contract rather than providing a clear error message about insufficient balance.

Recommendation

We recommend adding a balance check before creating the redeem message to validate that the contract has sufficient `stZIG` tokens. This should include a descriptive error message if the balance is insufficient, making debugging easier if this edge case ever occurs.

Status: Resolved