Audit Report

IBCX

v1.0

April 29, 2023

Table of Contents

Table of Contents **2**
License **4**
Disclaimer **4**
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' inactivity could permanently freeze the protocol 12
2. Rebalance finalization may become unachievable due to streaming fee collection 12
3. Inability to inflate an asset caused by using the wrong coin denomination for swapping the reserve to asset 13
4. Asset deflation uses wrong swap direction by swapping the reserve to the asset 13
5. Deflating an asset distributes the asset amount rather than the reserve amount 14
6. Total supply is not correctly computed 14
7. ClaimProof does not include beneficiary address, which allows front-running of airdrops 15
8. The same claim proof can be used multiple times 15
9. Inability to mint, burn and rebalance index tokens following the initial deflate rebalance trade 16
10. Rebalance may not be finalizable because of rounding error 16
11. Burning index tokens via the periphery contract yields substantially fewer asset tokens 17
12. The streaming fee calculation formula returns wrong results for some input values 18
13. Minting and burning index tokens interferes with rebalancing and can render rebalance finalization unachievable 18
14. Asset inflation simulates the swap incorrectly and expands by the wrong amount 19
15. An invalid rebalance configuration could prevent its finalization 20
16. Streaming fee realization mechanism can be manipulated by the fee collector to maximize profit 21
17. Tokens that are sent by mistake are not refunded when minting 21
18. Fees rates are not validated 22
19. Inconsistent query results and failed index token burns due to lack of considering a minimum fee for small index token amounts 22
20. Reserve token denom cannot be updated until the initial deflate rebalance trade is

completed 23
21. Idle rebalance manager can permanently disable rebalances 23
22. ContractError::SimulateQueryError uses incorrect input and output coin denominations 24
23. Centralization risks involved in performing governance-driven rebalancing 24
24. Contracts should implement a two-step ownership transfer 25
25. Unnecessary memory allocations 25
26. Multiple if statements make the code less readable 26
Appendix: Test Cases **27**
1. Test case for “Users' inactivity could permanently freeze the protocol” 27
2. Test case for “The streaming fee calculation formula returns wrong results for some input values” 28

License

THIS WORK IS LICENSED UNDER A CREATIVE COMMONS ATTRIBUTION-NODERIVATIVES 4.0 INTERNATIONAL 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.

COPYRIGHT OF THIS REPORT REMAINS WITH THE AUTHOR.

This audit has been performed by

Oak Security

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

Introduction

Purpose of This Report

Oak Security has been engaged by Osmosis Grants Company to perform a security audit of IBCX.

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/many-things/ibcx-contracts/
Commit	`d3b6ecf1d4a833897987a8d87b3ac86ab0f716f5`
Scope	All contracts were in scope.

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

IBCX is a Cosmos index token that is designed to track the performance of the leading tokens in the Cosmos ecosystem. It aims to procure and maintain those tokens as its collateral and rebalance them through governance voting.

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**, or **Resolved**.

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

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

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

Code Quality Criteria

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

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Low	No technical documentation was available.
Test coverage	Low-Medium	`cargo tarpaulin` reports 52.81% code coverage.

Summary of Findings

No	Description	Severity	Status
1	Users' inactivity could permanently freeze the protocol	Critical	Resolved
2	Rebalance finalization may become unachievable due to streaming fee collection	Critical	Resolved
3	Inability to inflate an asset caused by using the wrong coin denomination for swapping the reserve to asset	Critical	Resolved
4	Asset deflation uses wrong swap direction by swapping the reserve to the asset	Critical	Resolved
5	Deflating an asset distributes the asset amount rather than the reserve amount	Critical	Resolved
6	Total supply is not correctly computed	Critical	Resolved
7	`ClaimProof` does not include beneficiary address, which allows front-running of airdrops	Critical	Resolved
8	The same claim proof can be used multiple times	Critical	Resolved
9	Inability to mint, burn and rebalance index tokens following the initial deflate rebalance trade	Critical	Resolved
10	Rebalance may not be finalizable because of rounding error	Critical	Resolved
11	Burning index tokens via the periphery contract yields substantially fewer asset tokens	Critical	Resolved
12	The streaming fee calculation formula returns wrong results for some input values	Major	Resolved
13	Minting and burning index tokens interferes with rebalancing and can render rebalance finalization unachievable	Major	Resolved
14	Asset inflation simulates the swap incorrectly and expands by the wrong amount	Major	Resolved
15	An invalid rebalance configuration could prevent its finalization	Major	Resolved
16	Streaming fee realization mechanism is	Major	Resolved

	manipulable by the fee collector to maximize profit		
17	Tokens that are sent by mistake are not refunded when minting	Major	Resolved
18	Fees rates are not validated	Minor	Resolved
19	Inconsistent query results and failed index token burns due to lack of considering a minimum fee for small index token amounts	Minor	Resolved
20	Reserve token denom cannot be updated until the initial deflate rebalance trade is completed	Minor	Resolved
21	Idle rebalance manager can permanently disable rebalances	Minor	Resolved
22	`ContractError::SimulateQueryError` uses incorrect input and output coin denominations	Minor	Resolved
23	Centralization risks involved in performing governance-driven rebalancings instead of automated ones	Informational	Acknowledged
24	Contracts should implement a two-step ownership transfer	Informational	Resolved
25	Unnecessary memory allocations	Informational	Resolved
26	Multiple if statements make the code less readable	Informational	Resolved

Detailed Findings

1. Users' inactivity could permanently freeze the protocol

Severity: Critical

The `calculate_streaming_fee` function defined in `contracts/core/src/state/mod.rs:44-75` calculates the time-weighted streaming fee with the following formula:

$$rate = (1 + fee_{stream})^{\Delta time} - 1$$

where $\Delta time$ is the elapsed time since the latest stream fee application.

If this time delta gets too big, the exponential operation could perform an overflow, reverting the message handling. This could happen, for example, after a chain upgrade or during a bear market where users are less interested in interacting with the contract.

Since the `calculate_streaming_fee` function is executed for each `Execute` message, its overflow and revert of the message would permanently freeze the protocol.

A test case is provided in Appendix 1.

Recommendation

We recommend reworking the formula that calculates the time-weighted streaming fee in order to support high $\Delta time$ values. An alternative solution would be to offer a recovery entry point that can be used to iteratively calculate the streaming fee over multiple small time deltas.

Status: Resolved

2. Rebalance finalization may become unachievable due to streaming fee collection

Severity: Critical

Streaming fees are collected via the `collect_streaming_fee` function implemented in `contracts/core/src/execute/fee.rs` on every message execution in `contracts/core/src/contract.rs:67`, including rebalance messages.

If the deflation target units are attained during rebalancing (i.e., `target_unit == current_unit` – notice the strict equality check), decrementing the collected fee from the current asset units in line `139` of the `collect_streaming_fee` function will result in an inability to pass the strict equality check in the `finalize` function.

As a result, rebalancing becomes stuck and cannot be finalized.

Recommendation

We recommend refraining from collecting and realizing streaming fees during rebalancing.

Status: Resolved

3. Inability to inflate an asset caused by using the wrong coin denomination for swapping the reserve to asset

Severity: Critical

The `inflate` function incorrectly uses the `RESERVE_DENOM` constant as the `token_in` argument for the `msg_swap_exact_in` call in `contracts/core/src/execute/rebalance.rs:436`.

Since `RESERVE_DENOM` has a constant `"reserve"` value, which is not a valid coin denomination, the intended swap fails, causing the transaction to revert.

Recommendation

We recommend using the `token.reserve_denom` coin denomination as the `token_in` argument for the `msg_swap_exact_in` function to correctly inflate the asset by swapping the reserve to the asset.

Status: Resolved

4. Asset deflation uses wrong swap direction by swapping the reserve to the asset

Severity: Critical

The process of deflating an asset during rebalancing involves swapping an exact amount of the asset (`denom`) to the reserve (`token.reserve_denom`).

However, in the current implementation, the `deflate` function in `contracts/core/src/execute/rebalance.rs` uses the asset coin denomination `denom` as the `token_out` argument for the `msg_swap_exact_out` function, resulting in the reserve being swapped to the asset rather than the intended swap of the asset to the reserve.

Recommendation

We recommend using the `msg_swap_exact_in` function to swap an exact amount of the asset (`denom`) to the reserve (`token.reserve_denom`). Additionally, the

`sim_swap_exact_in` function should be used to determine the resulting amount of reserve tokens.

Furthermore, to ensure adequate slippage protection, we recommend changing the `max_amount_in` parameter of the `deflate` function to `min_amount_out`.

Status: Resolved

5. Deflating an asset distributes the asset amount rather than the reserve amount

Severity: Critical

The `deflate` function within `contracts/core/src/execute/rebalance.rs` is designed to distribute the reserve amount received to the `RESERVE_BUFFER` via the `distribute_after_deflate` function for subsequent asset inflation.

However, the current implementation mistakenly distributes the asset amount instead of the reserve amount. This can lead to incorrect accounting of reserves and prevent subsequent inflation of assets.

Recommendation

We recommend using the reserve amount to distribute to the buffer.

Status: Resolved

6. Total supply is not correctly computed

Severity: Critical

In `contracts/core/src/execute/mod.rs:100`, during the execution of `Burn` messages, the total supply is decreased by the amount of the received tokens.

However, since part of the user's provided tokens is sent to the `fee_collector` because of the burn fee, the computed total supply is not correct because it is not accounting for the fees.

This implies that the total supply recorded in the contract is smaller than the actual one and a `Burn` message from the `fee_collector` would redeem more collateral than expected.

Recommendation

We recommend subtracting the fee-deducted amount from the total supply.

Status: Resolved

7. `ClaimProof` does not include beneficiary address, which allows front-running of airdrops

Severity: Critical

Two types of proofs are supported for airdrops – in `contracts/airdrop/src/execute.rs:108-116`, the logic for extracting the `claim_proof` distinguishes between the `Account` and `ClaimProof` types. While the `Account` type always includes the address of the beneficiary, `ClaimProof` allows claiming with an arbitrary string.

This is problematic because it enables front-running of these claims. When an attacker sees the `proof` string, he can submit it himself and claim the amount before the user.

Recommendation

We recommend always including the beneficiary in the leaf.

Status: Resolved

8. The same claim proof can be used multiple times

Severity: Critical

In `contracts/airdrop/src/lib.rs:18`, the user input that is proven is defined as `{claim_proof}{amount}`, i.e., concatenation of the claim proof and the amount. This input is verified with the provided claim proof.

Because this encoding is ambiguous, an attacker can craft multiple combinations that result in the same user input and which are not rejected as invalid or duplicates.

For instance, let us assume that `claim_proof = abcd1234`. An amount of `5` results in a user input of `abcd12345`, which is the value of the leaf. However, setting `claim_proof = abcd123` and `amount = 45` also results in user input of `abcd12345`. Because it is only verified that the `claim_proof` was not used before (in `contracts/airdrop/src/execute.rs:142`), both calls will succeed and the user can claim `45 + 5 = 50` tokens instead of the intended `5`.

Recommendation

We recommend including a separator between `claim_proof` and `amount` when constructing the user input.

Status: Resolved

9. Inability to mint, burn and rebalance index tokens following the initial deflate rebalance trade

Severity: Critical

Deflating of an asset persists and updates the temporary accounting asset `RESERVE_DENOM` as part of the `UNITS` map alongside the other assets of the index. This `RESERVE_DENOM` is a constant string that resolves to `"reserve"` and is not a valid coin denomination. Any attempt to transfer this asset will result in an error.

Due to the assumption that all assets within the `UNITS` map are tokens with valid coin denominations, the core functionalities of the protocol - minting, burning, and rebalancing - stop working after the initial deflate rebalance trade, as `RESERVE_DENOM` is also added to the `UNITS` map.

Minting – Since minting index tokens requires sending all asset tokens of `UNITS` along with the minting message, it is not possible to mint index tokens as `RESERVE_DENOM` cannot be transferred.

Burning – The `get_redeem_amounts` function in `contracts/core/src/state/units.rs` determines the amount of underlying assets to redeem to the user when burning index tokens. Those assets are then transferred to the user. However, as `RESERVE_DENOM` is kept within the `UNITS` map and not removed from the redeemable assets, the transfer and hence the transaction reverts.

Rebalancing – In the case of streaming fees being configured, fees are automatically collected and realized on every rebalance message. However, fees are also incorrectly collected from `RESERVE_DENOM`. As it is not possible to transfer the collected fees of the `RESERVE_DENOM` coin denomination, rebalancing reverts.

Recommendation

We recommend storing the deflated reserve amount currently persisted as `RESERVE_DENOM` in the `UNITS` map as a separate value in storage.

Status: Resolved

10. Rebalance may not be finalizable because of rounding error

Severity: Critical

The function `distribute_after_deflate` distributes the amount provided after an inflation operation to the `RESERVE_BUFFER` entries for the different tokens. It does so by first calculating the individual share for each token by dividing the weight by the total weight and then multiplying this ratio by the amount.

However, because fixed point arithmetic is used, there can be a very small loss of precision (in the 18th decimal place) when performing the division. While this loss of precision generally

would be negligible, it breaks an important invariant of the rebalance operation: The sum of all `RESERVE_BUFFER` entries will no longer be equal to the `RESERVE_DENOM` balance.

This is problematic for the finalization. In `contracts/core/src/execute/rebalance.rs:470`, it is required that the `RESERVE_DENOM` balance is exactly zero. But since this balance can only be decreased by consuming `RESERVE_BUFFER` entries, the condition may fail when there is a small amount left due to the imprecision of the calculation. Therefore, finalizing the rebalance operation will not be possible.

Recommendation

We recommend increasing the `RESERVE_DENOM` balance by the sum of the distributed amount that is calculated in `distribute_after_deflate`. This ensures that `RESERVE_DENOM` always stays in sync with the sum of the `RESERVE_BUFFER` entries.

Status: Resolved

11. Burning index tokens via the periphery contract yields substantially fewer asset tokens

Severity: Critical

Burning index tokens via the periphery `BurnExactAmountIn` message invokes the `burn_exact_amount_in` function in `contracts/periphery/src/execute.rs`. This function calls the core `simulate_burn` function located in `contracts/core/src/query.rs` to determine the redeemable amounts of asset tokens, which are then swapped to a desired token (`output_asset`).

However, the `simulate_burn` function incorrectly applies the burn fee to `amount_with_fee`. Instead of calculating the burn fee amount and deducting it from the provided `amount` of index tokens, it multiplies the `amount` with the `fee.burn` ratio in line `119`. This results in the `amount_with_fee` being just the fee, not the amount minus the fee.

Consequently, the `amount_with_fee` is substantially smaller than the provided `amount` of index tokens. As a result, the periphery contract will swap only a fraction of the redeemed assets to the desired token.

As users can provide a minimum amount (`min_output`) of the desired token to receive, the transaction fails if this condition is not met.

This renders the `BurnExactAmountIn` periphery message unusable.

Recommendation

We recommend calculating the effective amount `amount_with_fee` of index tokens to burn by deducting the fee from the provided `amount` of index tokens.

Status: Resolved

12. The streaming fee calculation formula returns wrong results for some input values

Severity: Major

The `calculate_streaming_fee` function defined in `contracts/core/src/state/mod.rs:44-75` calculates the time-weighted streaming fee with the following formula:

$$rate = (1 + fee_{stream})^{\Delta time} - 1$$

where $\Delta time$ is the elapsed time from the latest stream fee application.

Since this function returns a rate that has to be deducted from token `units`, the result must be in the `[0,1]` range. Since the result depends on the $\Delta time$ and the fee_{stream} input variables, it may fall outside of this range for some of those values.

Results outside of the mentioned range would cause wrong calculations resulting in token `units` being incorrectly reduced and accrued to the fee collector.

A graphical simulation is provided in the Appendix.

Recommendation

We recommend reworking the formula that calculates the time-weighted streaming fee in order to always return a value in the `[0,1]` range.

Status: Resolved

13. Minting and burning index tokens interferes with rebalancing and can render rebalance finalization unachievable

Severity: Major

As minting and burning index tokens continue to work while rebalancing is ongoing, the total supply of index tokens can fluctuate during rebalancing.

During rebalancing, the `deflate_reserve` and `deflate` functions deflate assets, while the `inflate_reserve` and `inflate` functions inflate assets in

`contracts/core/src/execute/rebalance.rs`. These functions use the current supply of index tokens (`token.total_supply`) to perform calculations.

For example, the `inflate_reserve` function calculates the `swap_unit` ratio of the `amount` of reserve tokens to the total supply of index tokens (`token.total_supply`).

The reserve unit `RESERVE_DENOM` is then reduced by `swap_unit` in `contracts/core/src/execute/rebalance.rs:355` and persisted in the `UNITS` map.

As the `token.total_supply` changes during the rebalancing process, the `swap_unit` increases or decreases accordingly. This can result in `RESERVE_DENOM` never reaching `0`, which is required to finalize the ongoing rebase.

Recommendation

We recommend temporarily pausing the minting and burning of index tokens during the rebalancing process, similarly to how other protocols handle rebalancing.

Status: Resolved

14. Asset inflation simulates the swap incorrectly and expands by the wrong amount

Severity: Major

Inflating an asset by a certain amount is achieved by swapping the reserve to the asset.

To determine the resulting amount (`amount_out`) of the asset (`denom`), the token swap is simulated using the `sim_swap_exact_in` function in `contracts/core/src/execute/rebalance.rs:402`.

However, instead of using the reserve coin denomination (`token.reserve_denom`), the asset `denom` is used as the `token_in` argument for the `sim_swap_exact_in` function. As a result, the swap is simulated from the asset to the reserve instead of the reserve to the asset.

This results in either a failed transaction due to the unsatisfied `min_amount_out` slippage protection or, if the slippage protection is bypassed, in using the wrong amount (`amount_out`) to calculate the expansion amount (`expand_unit`) of the asset.

Additionally, this incorrect `amount_out` value is supplied as the `token_out_min` argument for the `msg_swap_exact_in` function in line `438`, potentially causing the swap to fail due to the unsatisfied slippage protection condition.

Recommendation

We recommend using the `token.reserve_denom` coin denomination as the `token_in` argument for the `sim_swap_exact_in` function to correctly determine the amount of `denom` tokens (`amount_out`) to inflate by.

Status: Resolved

15. An invalid rebalance configuration could prevent its finalization

Severity: Major

The `Rebalance::validate` function in `contracts/core/src/state/rebalance.rs` is responsible for validating the rebalance config before the protocol's governance can initiate a new rebalance. Once a rebalance has started, the config can not be altered until the rebalance is finalized.

However, the current implementation of the `Rebalance::validate` function is missing some important checks, which can lead to failed rebalances.

Specifically, if the inflation ratios in the configuration are incomplete or empty, it will not be possible to utilize the reserve fully. When attempting to finalize the rebalance, the non-zero check of `UNITS[RESERVE_DENOM]` in `contracts/core/src/execute/rebalance.rs:468` will fail, causing the finalization of the rebalance to fail.

Similarly, if the configuration includes duplicate coin denominations within the `inflation` and `deflation` vectors, the finalization of the rebalance will fail because of the guard in `contracts/core/src/state/rebalance.rs:456-463`. This results in the impossibility of a `Rebalance` being finalized as strict equality cannot be true for two instances of the same denom with different amounts.

We classify this issue as major because only governance can initiate a rebalance and define its configuration.

Recommendation

We recommend adding additional checks to the `validate` function to ensure the rebalance configuration is valid and can not lead to a stuck rebalance.

Status: Resolved

16. Streaming fee realization mechanism can be manipulated by the fee collector to maximize profit

Severity: Major

The function `realize_streaming_fee` can be executed by the designated fee collector address at any time in order to collect fees.

The collected fees are calculated by multiplying the `unit` value with the `token.total_supply` in `contracts/core/src/execute/fee.rs:156` for each `Coin`.

This is problematic because while `unit` is a time-weighted value, `token.total_supply` is a value that fluctuates over time, leading to too high or low streaming fees.

Moreover, it may be economically feasible for the fee collector to temporarily inflate the total supply (by minting), execute `realize_streaming_fee`, and burn the tokens again.

Recommendation

We recommend using the average total supply over the time period since the last realization.

Status: Resolved

17. Tokens that are sent by mistake are not refunded when minting

Severity: Major

The function `mint` calls `assert_units` to calculate how many tokens to refund for all backing assets in `contracts/core/src/execute/mod.rs:43`. The `assert_units` function checks that `info.funds` contains an entry for every element in `assets`, in order to calculate refunds for these elements. However, it is not validated that `info.funds` does not contain any additional entries. Therefore, `info.funds` is checked to be a superset of `assets`.

This means that if a user sends other tokens along the call, those will be lost and not refunded.

Recommendation

We recommend returning an error when the user sends assets other than the expected ones.

Status: Resolved

18. Fees rates are not validated

Severity: Minor

In `contracts/core/src/contract.rs:30-40`, during the contract's instantiation, input provided fee rates defined in `msg.fee_strategy` are not validated to be in the `[0,1]` range.

Values outside of that range will cause errors in calculations since fees could exceed amounts.

Recommendation

We recommend validating fee rate values to be in the `[0,1]` range.

Status: Resolved

19. Inconsistent query results and failed index token burns due to lack of considering a minimum fee for small index token amounts

Severity: Minor

The `simulate_mint` and `simulate_burn` functions in `contracts/core/src/query.rs` lack the application of a minimum fee of `Uint128::one()`, unlike the `make_mint_msgs_with_fee_collection` and `make_burn_msgs_with_fee_collection` functions in `contracts/core/src/execute/fee.rs`.

This absence of a minimum fee in the `simulate_mint` function results in incorrect query results.

Additionally, the lack of applying a minimum fee in the `simulate_burn` function results in a failed burn transaction for small index token amounts (if `amount * fee < 1`) when used via the periphery contract. This occurs because the `make_burn_swap_msgs` function in `contracts/periphery/src/msgs.rs` expects a larger amount of redeemed asset tokens than actually received and fails to swap due to insufficient funds.

Recommendation

We recommend considering the same minimum fee of `Uint128::one()` in both the `simulate_mint` and `simulate_burn` functions.

Status: Resolved

20. Reserve token denom cannot be updated until the initial deflate rebalance trade is completed

Severity: Minor

The current reserve coin denomination is stored in the `reserve_denom` field of the `TOKEN` struct and can be updated through governance with the `update_reserve_denom` function in `contracts/core/src/execute/gov.rs:142`.

When attempting to load the unit with the `RESERVE_DENOM` key using the `cw-storage-plus` `Map::load` function, an error will occur if the key does not exist yet. As the `RESERVE_DENOM` key is only set after the first deflate rebalance trade is complete, it is not possible to change the reserve token denomination until then.

Recommendation

We recommend using the `Map::may_load` function to load the `RESERVE_DENOM` key from storage if it has not been set yet.

Status: Resolved

21. Idle rebalance manager can permanently disable rebalances

Severity: Minor

When the governance sends an `Init` message in order to initialize a rebalance, they elect a `manager` that has the right to perform trades on their behalf in order to match the rebalance requirements.

However, since only one rebalancing operation can be in the active state and there is no deadline defined for the `manager` to execute trades, an idle `manager` can permanently disable rebalances.

We classify this issue as minor since the manager is a trusted party elected through governance.

Recommendation

We recommend defining a deadline for the manager to execute trades and after that enable anyone to finalize the rebalance.

Status: Resolved

22. `ContractError::SimulateQueryError` uses incorrect `input` and `output` coin denominations

Severity: Minor

The `make_burn_swap_msgs` function in `contracts/periphery/src/msgs.rs` builds the swap messages required for swapping the received underlying assets to a desired output token (`min_output`).

The minimum output token amount is determined by simulating the swap via the `sim_swap_exact_in` function. While the arguments for the simulation are correct, the `input` and `output` coin denominations supplied to the `ContractError::SimulateQueryError` error in lines `104` and `105` are flipped and lead to an incorrect error message.

Recommendation

We recommend providing `denom` as the `input` and `min_output.denom` as the `output` coin denominations to the `ContractError::SimulateQueryError` error.

Status: Resolved

23. Centralization risks involved in performing governance-driven rebalancing

Severity: Informational

The protocol is designed to rely on governance inputs to decide the composition and ratios of the token's collaterals.

In fact, rebalancing can be initiated only by governance, which also provides instructions for token inflation and deflation that will be executed arbitrarily by a third-party manager.

This implies that the rebalancing process is not automated, can take time, and generate divisions in the community since no algorithm is specified. It also implies that governance is the single point of failure of the protocol.

Other protocols use automated rule-based rebalance systems that take into account objective metrics such as market capitalization in order to define the list of collateral assets.

Recommendation

We recommend automating or partially automating token rebalancing as well as allowing anyone to initiate rebalancing in a permissionless way. Governance could vote on on-chain rebalancing strategies. This would keep the system automated and transparent, but it would still be possible to react to changing market conditions by changing strategies.

Status: Acknowledged

24. 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 the 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

25. Unnecessary memory allocations

Severity: Informational

The codebase presents opportunities to reduce the amount of memory allocations:

- In `contracts/periphery/src/msg.rs:37` is an unnecessary clone.
- In `contracts/core/src/state/rebalance.rs:80`, a vector allocation can be prevented when `f.is_empty()` in line `80`.

Recommendation

We recommend:

- Removing the `clone` in `contracts/periphery/src/msg.rs:37`.
- Not allocating the vector in `contracts/core/src/state/rebalance.rs:79`, and working with the iterator instead to verify whether the array is empty by using the `Iterator::count` method.

Status: Resolved

26. Multiple if statements make the code less readable

Severity: Informational

In `contracts/core/src/state/mod.rs:44`, the `calculate_streaming_fee` function contains two nested if statements that check whether there is a `stream` available and that the `elapsed` time is bigger than 0. If those two conditions are met, the function updates both the `units` and `fee`.

In Rust, it is possible to leverage the match keyword to merge these two branches into one and flatten the code, hence improving readability.

Recommendation

We recommend matching on `self.stream`, and using the condition `now - self.stream_last_collected_at > 0` for one path and handle all other cases on the other one. This will reduce one level of depth and remove one conditional branch.

Status: Resolved

Appendix: Test Cases

1. **Test case for [“Users' inactivity could permanently freeze the protocol”]**

The following test case executes the formula with the following parameters in order to cause an overflow:

- fee_{stream} = 5%
- $\Delta time$ = 1200 seconds (20 minutes)

```
#[test]
    fn inactivity_test() {

        // 5% stream fee
        let stream_fee = Decimal::from_str("0.05").unwrap();


        // 5 seconds (1 block) elapsed
        let elapsed = 5;
        let rate = (Decimal::one() + stream_fee)
        .checked_pow(elapsed as u32).unwrap()
        .checked_sub(Decimal::one()).unwrap();

        dbg!(rate, rate.to_string());

        // 20 minutes elapsed
        let elapsed = 1200;
        let rate = (Decimal::one() + stream_fee)
        .checked_pow(elapsed as u32).unwrap() // Overflow error
        .checked_sub(Decimal::one()).unwrap();

        dbg!(rate, rate.to_string());
    }
```

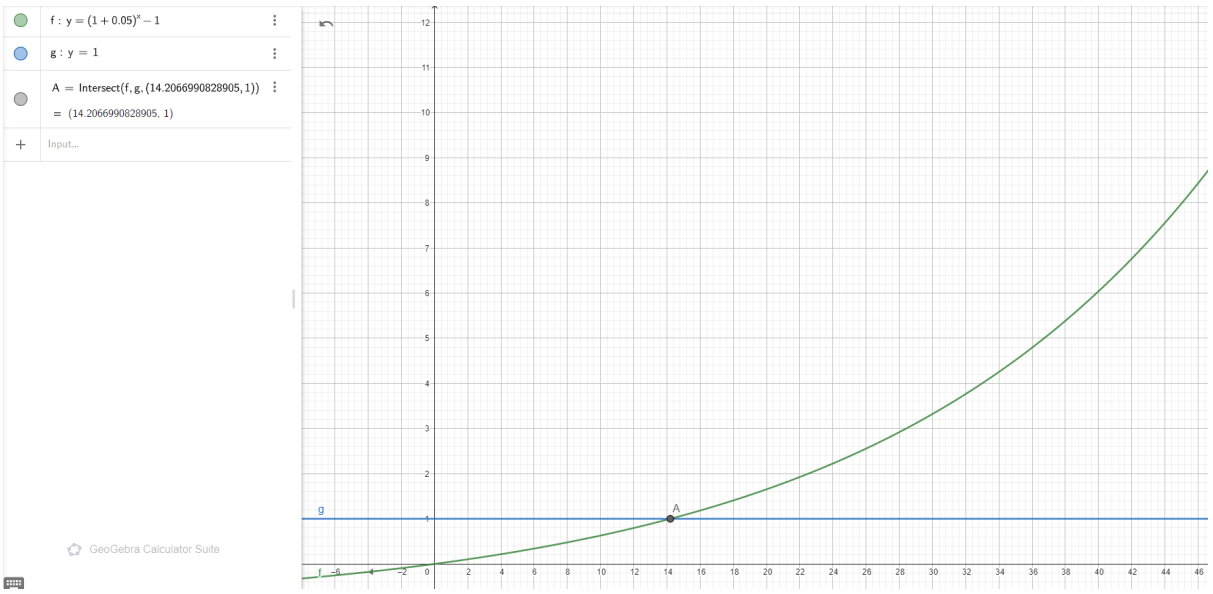
2. Test case for [“The streaming fee calculation formula returns wrong results for some input values”]

```
#[test]
    fn inactivity_test() {

        // 5% stream fee
        let stream_fee = Decimal::from_str("0.05").unwrap();


        // 2 minutes elapsed
        let elapsed = 120;
        let rate = (Decimal::one() + stream_fee)
        .checked_pow(elapsed as u32).unwrap()
        .checked_sub(Decimal::one()).unwrap();

        dbg!(rate, rate.to_string());
    }
```

```
[contracts/core/src/state/mod.rs:193] rate = Decimal(
    Uint128(
        347911985667201628364,
    ),
)
[contracts/core/src/state/mod.rs:193] rate.to_string() =
"347.911985667201628364"
```

The following graphs draw the streaming fee formula $\Delta time$ dependent.

This first graph is parametrized with fee_{stream} value of 5% and shows how the result diverges from the `[0,1]` range if the $\Delta time$ is outside the `[0,14.29]` range.



By decreasing the fee_{stream} value to 0.5%, the function returns meaningful values with $\Delta time$ in the `[0,138.97]` range.