Audit Report

SendIt Contracts

v1.0

September 25, 2024

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 Neutron Audit Sponsorship Program to perform a security audit of SendIt 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/sendit-zone/cosmwasm-contracts
Commit	`b0cb90b640497677a26304a901a61aa5365b2e6c`
Scope	The scope was restricted to the following directories: • `contracts/bl-market` • `contracts/liquidation-queue` • `contracts/p-token` • `contracts/oracle` • `packages/bl-core`
Fixes verified at commit	`4476c6abc1f39d3c3b4dc3d975f7707a17360238` 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

SendIt is a money market lending protocol that implements a fully automated on-chain credit facility governed by a decentralized community.

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	Medium-High	-
Test coverage	Medium-High	`cargo-llvm-cov` reports the following coverage: • `78.21%` function coverage • `90.77%` line coverage • `61.84%` region coverage

Summary of Findings

No	Description	Severity	Status
1	Liquidation queue contract receives fewer collaterals than expected when liquidating more than one asset	Critical	Resolved
2	Users can steal funds by receiving more liquidation fees, causing a loss of funds for bidders	Critical	Resolved
3	Utilization rate can be exploited to surpass 100% for new markets	Critical	Resolved
4	Incorrect price mechanism used during liquidation	Critical	Resolved
5	Users can borrow funds for free due to rounding errors	Critical	Resolved
6	Liquidation failure due to incorrect address validation	Critical	Resolved
7	Borrowed asset bit is not removed after liquidation, causing withdrawals to fail	Critical	Resolved
8	Hardcoded stablecoin peg may compromise computations	Critical	Resolved
9	Refund amount is not excluded when computing the utilization ratio	Major	Resolved
10	Deactivated markets can be liquidated and repaid	Major	Resolved
11	Risk of withdrawing funds within the liquidation threshold limit	Major	Resolved
12	Liquidation may fail due to out-of-gas errors	Major	Resolved
13	Market deposit limits may be surpassed	Major	Resolved
14	Premium slot validation is voided	Major	Resolved
15	Free and instant bid retractions create risk-free profit vectors for well-funded attackers	Major	Resolved
16	Linear premium slots might lead to suboptimal risk of liquidator incentives	Minor	Acknowledged
17	Missing validation when updating configurations	Minor	Resolved

18	Non-existing price sources can be removed	Informational	Resolved
19	Lack of a secondary fallback price feed oracle	Informational	Acknowledged
20	Volatile assets may be configured with insufficient thresholds	Informational	Acknowledged
21	Denom validation can be simplified	Informational	Resolved
22	Users cannot withdraw assets when the market is deactivated	Informational	Resolved
23	Unhandled division by zero error	Informational	Resolved
24	Missing validations for liquidation price queries	Informational	Resolved
25	Miscellaneous comments	Informational	Resolved
26	Fixed price oracles can be set to zero	Informational	Resolved
27	Usage of magic numbers decreases maintainability	Informational	Resolved

Detailed Findings

1. Liquidation queue contract receives fewer collaterals than expected when liquidating more than one asset

Severity: Critical

In `contracts/bl-market/src/execute.rs:1211-1286`, the `liquidate` function implements logic to execute a liquidation on unhealthy positions. Based on the `LiquidationAmountResponse` queried from the liquidation queue contract, the `process_ptoken_transfer_to_liquidation_contract` function will be called for each collateral to transfer pTokens from the borrower to the liquidation queue contract. These messages are intended to be collected within the `Response` created in line `1209`.

However, the `Response` variable will be reinitialized in lines `1264-1271` when there is more than one collateral to liquidate, causing the previous collateral's `TransferOnLiquidation` message not to be dispatched. This is incorrect because all the collaterals must be transferred to the liquidation queue contract as part of the liquidation logic.

Consequently, the liquidation queue contract will only receive one out of all the expected collaterals, causing a loss of funds for bidders as there are insufficient collaterals to be claimed.

Please see the `test_liquidate_double_asset_poc` test case in the appendix to reproduce this issue.

Recommendation

We recommend accumulating the `TransferOnLiquidation` sub-messages by modifying the `process_ptoken_transfer_to_liquidation_contract` function to accept a `Vec<SubMsg>` argument, which is added to the method's returned `Response` via `add_submessages`. When this method is called, the parameter should be provided with `response.messages.clone()`.

Status: Resolved

2. Users can steal funds by receiving more liquidation fees, causing a loss of funds for bidders

Severity: Critical

In `contracts/liquidation-queue/src/contract.rs:1074`, the `execute_dynamic_liquidation` function allows the caller to specify the `collateral_amount` parameter, representing the amount of collateral tokens to liquidate. In return, the caller will receive a portion of the repayment amount based on the

`config.liquidator_fee` percentage, as seen in `contracts/liquidation-queue/src/contract.rs:1197-1205`.

When the `LiquidateDirect` message is called in `contracts/liquidation-queue/src/contract.rs:1172-1184`, the `calculate_liquidation_amounts` function computes the required repayment amount based on the borrower's debt position. If any excess debt token is sent, the tokens will be refunded in `contracts/bl-market/src/liquidate.rs:203-211`.

However, the `liquidation-queue` contract does not account for this scenario, allowing users to receive more liquidation fees than intended because the provided collateral amount is not fully consumed during liquidation. The contract will receive a lower collateral amount than expected, resulting in an insufficient funds error when bidders call the `ClaimLiquidations` message.

Consequently, malicious users can receive more liquidation fees by specifying the `collateral_amount` parameter to a value higher than the borrower's collateral balance, causing refunded tokens to be stuck and risking the protocol's solvency.

Recommendation

We recommend only distributing the fees and updating the state changes based on the repayment amount. The correct repayment amount can be computed by checking the difference of the `liquidation-queue` contract's balance before and after the liquidation.

Status: Resolved

3. Utilization rate can be exploited to surpass 100% for new markets

Severity: Critical

In `contracts/bl-market/src/interest_rate.rs:279-281`, the utilization rate of a market is determined by the ratio of total debt divided by total collateral. If a new market is instantiated with zero deposits, an attacker can inflate the utilization rate to steal funds from the contract.

An exemplary step-by-step attack follows:

1. `ATOM` market is instantiated.
2. The attacker creates two accounts: `account1` and `account2`.
3. `account1` becomes the first depositor and deposits `1 uatom`, increasing the market's total collateral.
4. `account1` donates `1000 ATOM` to the contract directly.
5. `account2` deposits `10000 OSMO` as collateral.
6. `account2` borrows the donated `1000 ATOM`, increasing the market's total debt.
7. The current utilization rate will exceed `100%` because the total debt is higher than the total collateral.

8. `account1` receives around `84000 ATOM` after 10 seconds due to the inflated utilization rate.

For more information on this type of attack, please refer to the Silo Finance Vulnerability Disclosure.

Recommendation

We recommend limiting the utilization rate to a maximum value of 100% with the `min` function.

Status: Resolved

4. Incorrect price mechanism used during liquidation

Severity: Critical

When querying an asset price in `contracts/oracle/src/contract.rs:165-170`, the `ActionKind` parameter can be set to `Default` or `Liquidation`. Both options provide different prices based on the actions performed, whether it is a regular operation (e.g., check LTV when borrowing funds) or performing a liquidation, as indicated in the comment in `packages/bl-core/src/oracle.rs:44`.

If the action performed is liquidating a borrower, the price source should be queried with `ActionKind::Liquidation` to use the correct pricing. However, this is not enforced in the `Liquidate` and `LiquidateDirect` messages in `contracts/bl-market/src/execute.rs:1123` and `contracts/bl-market/src/liquidate.rs:61`, as the `query_price` function queries the price as `ActionKind::Default` in `contracts/bl-market/src/helpers.rs:326`.

Consequently, an inconsistency arises between liquidated collaterals and debt repayments in the `bl-market` and `liquidation-queue` contracts. For example, if the liquidation price source is configured higher than the default price source, the borrower's collateral amount computed in `contracts/bl-market/src/execute.rs:1264-1271` will be less than the actual amount liquidated in `contracts/liquidation-queue/src/contract.rs:996`, causing a loss of funds for the last bidder when they call the `ClaimLiquidations` message.

Recommendation

We recommend querying the price as `ActionKind::Liquidation` for the `Liquidate` and `LiquidateDirect` messages in the `bl-market` contract.

Status: Resolved

5. Users can borrow funds for free due to rounding errors

Severity: Critical

In `contracts/bl-market/src/execute.rs:782-789`, the `borrow` function calculates the borrow value and ensures that the user's total debt does not exceed the maximum debt limit. This is problematic because a rounding issue may occur when multiplying the borrowed amount by the debt token price, causing the borrowed value to round down to zero. For example, borrowing 1 `uosmo` at a price of $0.40 results in a borrow value of 0 (1 * $0.40 = 0.4) due to integer truncation.

Consequently, users can borrow funds without needing to maintain a collateralized position, causing a loss of funds for lenders and risking the protocol's solvency.

Recommendation

We recommend applying the `ceil` function when calculating the borrowed value.

Status: Resolved

6. Liquidation failure due to incorrect address validation

Severity: Critical

In `contracts/liquidation-queue/src/contract.rs:1578`, the `compute_collateral_weights` function performs an `addr_validate` validation on the asset label to ensure the supplied collateral is a CW20 token address. This is problematic because an asset may be configured as a native token denom, as seen in `contracts/bl-market/src/execute.rs:1165-1167`.

Consequently, calling the `LiquidationAmount` query in `contracts/bl-market/src/execute.rs:1198-1207` will always fail, preventing borrowers with undercollateralized positions from being liquidated, ultimately risking the protocol's solvency.

We classify this issue as critical because the documentation states that the assets to be supported on the market are ATOM, NTRN, and OSMO native tokens.

Recommendation

We recommend removing the `addr_validate` validation in `contracts/liquidation-queue/src/contract.rs:1578`.

Status: Resolved

7. Borrowed asset bit is not removed after liquidation, causing withdrawals to fail

Severity: Critical

In `contracts/bl-market/src/liquidate.rs:163-164`, the `liquidate` function does not call `unset_bit` to remove the borrowed asset from the `User` struct after repaying the borrower's debt. This is problematic because a division by zero error would occur in `contracts/bl-market/src/execute.rs:641-644` when withdrawing collaterals in a zero-debt position, which is incorrect as withdrawals should be permitted for non-borrowing positions (see `contracts/bl-market/src/execute.rs:615-616`).

For comparison, the `repay` function removes the borrowed asset in `contracts/bl-market/src/execute.rs:975-977`.

Recommendation

We recommend removing the borrowed asset from the `User` struct when the borrower's debt is fully repaid.

Status: Resolved

8. Hardcoded stablecoin peg may compromise computations

Severity: Critical

In `contracts/bl-market/src/helpers.rs:319-320`, the price of the `usd` denominated native asset is hardcoded to `Decimal::one()`. Hardcoding an exchange rate to a stablecoin may compromise computations, as stablecoins typically fluctuate around their peg during normal market conditions but may depeg during periods of distress.

Consequently, attackers might exploit this to steal funds with compromised computations or hedge themselves at the cost of the protocol's solvency. For example, the attacker purchases a depegged stablecoin on the open market and leverages it to borrow other assets, causing a loss of funds for lenders.

Recommendation

We recommend querying prices for stablecoins instead of hardcoding them.

Status: Resolved

9. Refund amount is not excluded when computing the utilization ratio

Severity: Major

In `contracts/bl-market/src/execute.rs:967`, the `repay` function calls `update_interest_rates` with the `liquidity_taken` argument set to zero. This is incorrect because the `liquidity_taken` argument should be set to the refund amount in `contracts/bl-market/src/execute.rs:938-946`.

The `update_interest_rates` function computes the utilization ratio based on the ratio of total debt to total collateral in `contracts/bl-market/src/interest_rate.rs:267-284`. Since a refund will be performed, the total collateral must be decreased by the refund amount to correctly compute the market's borrow rate and liquidity rate, as seen in `contracts/bl-market/src/interest_rate.rs:304-315`.

Consequently, the utilization ratio will be computed to be lower than intended, causing lenders to receive fewer rewards and borrowers to pay lesser interest.

Recommendation

We recommend setting the `liquidity_taken` argument to `refund_amount` in `contracts/bl-market/src/execute.rs:967`.

Status: Resolved

10. Deactivated markets can be liquidated and repaid

Severity: Major

In `contracts/bl-market/src/contract.rs:193`, the `LiquidateDirect` message does not return an error if the debt or collateral market is inactive. This violates the comment in `packages/bl-core/src/bl_market/mod.rs:104-105`, which indicates that a deactivated market cannot perform any actions, including debt repayment and liquidation.

Recommendation

We recommend returning an error if the debt or collateral market is inactive, similar to `contracts/bl-market/src/execute.rs:881-883` and `1111-1115`.

Status: Resolved

11. Risk of withdrawing funds within the liquidation threshold limit

Severity: Major

In `contracts/bl-market/src/execute.rs:638-644`, the `withdraw` function allows users to withdraw assets as long the liquidation threshold is not exceeded. This is incorrect because the function should instead check the user's maximum Loan-to-Value (LTV) restriction. The LTV is meant to be a safe buffer before the liquidation threshold is triggered so that users are not being liquidated after withdrawing their funds.

For comparison, the `borrow` function enforces the user's position to not exceed the max LTV restriction, as seen in `contracts/bl-market/src/execute.rs:784-789`.

Consequently, a borrower may borrow collaterals at the max LTV and withdraw the collateral until they are at the liquidation threshold, exposing them to liquidation risks and circumventing the risk management buffer of the protocol. In a strongly downward trending market, aggravated limited liquidity or block space for liquidations, this issue could lead to a steep increase in unhealthy positions, risking the protocol's solvency.

Recommendation

We recommend modifying the withdrawal function to check for the user's max LTV instead of the liquidation threshold.

Status: Resolved

12. Liquidation may fail due to out-of-gas errors

Severity: Major

In `contracts/bl-market/src/execute.rs:1123-1179`, multiple iterations occur over the `user_asset_position` vector, which may cause the transaction to fail due to an out-of-gas error. Specifically, the assets in the user's position are iterated in lines `1123`, `1143`, and `1156`.

Consequently, borrowers could exploit this by opening many asset positions to force liquidations to fail due to gas limits, causing a loss of funds for lenders and risking the protocol's solvency.

We classify this issue as major because the maximum number of positions that can be opened depends on the number of assets supported by the contract owner, which is a privileged account.

Recommendation

We recommend refactoring the code to limit iterations to a bounded set of user positions. Alternatively, consider limiting the number of assets the contract owner can configure to a lower value.

Status: Resolved

13. Market deposit limits may be surpassed

Severity: Major

In `contracts/bl-market/src/execute.rs:510-519`, the `deposit` function implements an incorrect method of validating that the total deposits do not exceed a market's deposit cap. Specifically, line `514` enforces the following condition:

```
market.deposit_cap_scaled <= market.ptoken_address.total_supply + deposit_amount * SCALING_FACTOR
```

This is problematic because the `get_scaled_liquidity_amount` function in lines `527-528` computes the mint amount based on the market's liquidity index. If the index increases, the minted pTokens amount will be lesser, as seen in `contracts/bl-market/src/interest_rate.rs:226-230`.

Consequently, the same amount of collateral deposited will result in different amounts of pTokens minted when the liquidity index exceeds `Decimal::one()`, resulting in the pToken's total supply to grow at a slower pace than the collateral deposits, ultimately causing the maximum deposit limit to be surpassed.

Please see the `test_deposit_limit_is_broken_poc` test case in the appendix to reproduce this issue.

Recommendation

We recommend modifying the implementation to track the total amount deposited and withdrawn instead of querying the total supply.

Status: Resolved

14. Premium slot validation is voided

Severity: Major

In `contracts/liquidation-queue/src/contract.rs:546-548`, the `submit_bid` function does not return an error if the provided premium slot is invalid. This is because `StdError` is simply instantiated but not returned.

Consequently, users can submit bids with the premium slot greater than the maximum slot, causing their bids not to be utilized for liquidation in line `980`, ultimately risking the protocol's solvency as the `liquidation-queue` contract fails to liquidate undercollateralized positions from the `bl-market` contract.

Recommendation

We recommend returning the error to prevent users from submitting invalid bids.

Status: Resolved

15. Free and instant bid retractions create risk-free profit vectors for well-funded attackers

Severity: Major

Allowing users to retract bids instantly in `contracts/liquidation-queue/src/contract.rs:836` generates the possibility for well-capitalized actors to place "spoof bids" to fill up the 0% discount slot to discourage other users from depositing into any bid pool.

This may allow attackers with big capital to front-run liquidations by lowering their bids from a 0% discount to the lowest possible slot in which their capital will be utilized, effectively increasing the price discount received for minimal operational gas cost.

Consequently, users with small capital will be disincentivized from depositing funds to the `liquidation-queue` contract as they will receive zero profit, allowing the attacker to receive most of the liquidation profits. This risks the protocol's solvency as the attacker has the majority control over the liquidation bidding mechanism. For example, if the attacker decides to retract all their bids during a market crash, borrowers with undercollateralized positions will not be liquidated on time, causing a loss of funds for lenders.

Recommendation

We recommend implementing a two-step retraction process, where users must request their retraction at time `T` and can only execute the withdrawal after buffer `x` has passed at time `T + x`.

Specifically, the protocol should consider all requested retractions whose execution timestamp must have elapsed before removing them from the bid pool, regardless of the

effective withdrawal of funds. This avoids introducing the same issue where bids are preemptively requested for retractions but are only executed when the conditions described above occur.

Status: Resolved

16. Linear premium slots might lead to suboptimal risk of liquidator incentives

Severity: Minor

The premium slots of the liquidation queue contract that discount the price of the collateral of liquidatable positions are configured linearly. This is problematic because it limits the customizability of the protocol and liquidation queue – particularly for larger positions of downward-trending collateral.

Typically, one would assume that for relatively stable assets/collateral, very small steps would already incentivize participation from liquidators. In contrast, larger steps may be needed to incentivize participation for more volatile collaterals. However, when collateral depegs or the overall market sentiment changes, it transitions from a stable to an unstable market.

A linear premium slot design is insufficient for such scenarios and assets because it would require smaller and larger steps. The first slots may contain small discounts, while those closer to the maximum could contain larger discounts for collateral under heavy distress.

Recommendation

We recommend implementing more flexible premium slots by allowing them to be fully customizable or increasing the discount factor per slot exponentially.

Status: Acknowledged

The client states that historical data mostly shows liquidations occuring at an average of 5%, but they can plan addressing this issue in future versions.

17. Missing validation when updating configurations

Severity: Minor

The following instances illustrate missing validation in the codebase:

- The `liquidation_bonus` parameter in `contracts/bl-market/src/execute.rs:276` is not validated as in line `396` to ensure the values are within the logical range, which may cause the `LiquidateDirect` message to fail.

- The `CONFIG` struct in `contracts/liquidation-queue/src/contract.rs:53` is not validated as in line `483`, which may cause invalid market and oracle timeframes to be configured.
- The `target_health_factor` parameter is not validated to be greater than or equal to 100% in `packages/bl-core/src/bl_market/mod.rs:118`, which may cause the borrower to be liquidated more than once.
- The `protocol_liquidation_fee` parameter is not validated to be lower than 100% in `packages/bl-core/src/bl_market/mod.rs:118`, which may cause the `LiquidateDirect` message to fail.

We classify this issue as minor because it can only be caused by a misconfiguration from the contract owner, which is a privileged address.

Recommendation

We recommend applying the recommendations above.

Status: Resolved

18. Non-existing price sources can be removed

Severity: Informational

In `contracts/oracle/src/contract.rs:150`, the `remove_price_source` function does not ensure the provided `denom` exists in the `PRICE_SOURCES` state before removing it. This would result in a no-op operation instead of returning an error because the `remove` function does not check whether the key exists in the storage, potentially confusing users as they believe the denom is a valid key inside the `PRICE_SOURCES` map.

Recommendation

We recommend implementing the `PRICE_SOURCES.has` validation to ensure the denom exists in the state before being removed.

Status: Resolved

19. Lack of a secondary fallback price feed oracle

Severity: Informational

The codebase depends on a single oracle contract with no backup oracles implemented. If the oracle's price is stale, the protocol operations will be incorrect and eventually suspended. It is best practice to configure a backup oracle.

Recommendation

We recommend implementing a secondary oracle.

Status: Acknowledged

The client states that this is a risk management decision and they can address this issue in future versions.

20. Volatile assets may be configured with insufficient thresholds

Severity: Informational

The health factor formula implemented resembles Aave. From the documentation, the Loan-to-Value (LTV) and liquidation threshold configurations differ based on the volatility of the assets. For example, less volatile assets typically have an 80% liquidation threshold and LTV values, while highly volatile assets have a 65% liquidation threshold and 35% LTV values.

However, this is not enforced within the codebase.

Recommendation

We recommend validating that the LTV and liquidation threshold parameters are within 25% to 90% ranges.

Status: Acknowledged

The client states that this is a risk management decision.

21. Denom validation can be simplified

Severity: Informational

In `contracts/bl-market/src/helpers.rs:161`, the `get_denom_amount_from_coins` function ensures that the user sends the required denom and only one coin. However, this check is performed using a non-standard approach.

Recommendation

We recommend using the cw_utils::must_pay utility library instead.

Status: Resolved

22. Users cannot withdraw assets when the market is deactivated

Severity: Informational

In `contracts/bl-market/src/execute.rs:562-564`, the `withdraw` function disallows users to withdraw their assets if the market is deactivated.

It is generally encouraged to allow users to exit the protocol during a market pause, especially when their assets are not used as collaterals. In the event of an emergency market pause, users may wish to protect their funds by withdrawing them, which is currently not possible.

Recommendation

We recommend allowing users to withdraw assets when the market is deactivated.

Status: Resolved

23. Unhandled division by zero error

Severity: Informational

In `contracts/liquidation-queue/src/contract.rs:1481`, a division by zero error would occur if the `total_weight` is zero. It is a best practice to prevent a division by zero by handling such cases with an error message.

Recommendation

We recommend returning an error if `total_weight` equals zero.

Status: Resolved

24. Missing validations for liquidation price queries

Severity: Informational

The protocol implements two modes for oracle price queries in `packages/bl-core/src/oracle.rs:46-49`:

- `Default`
- `Liquidation`

The default mode calculates a time-weighted average price and checks if the price is within a reasonable range in `contracts/oracle/src/pyth.rs:88-89`.

However, these validations are not implemented in the `query_pyth_price_for_liquidation` function in `contracts/oracle/src/pyth.rs:101`.

While the protocol implemented this as a design choice to enable liquidations at all times, it can be good risk management to enforce less strict requirements for liquidations to account for times of distress.

Recommendation

We recommend implementing a wider confidence interval for liquidations or warning borrowers that they are not protected against manipulation for liquidation price queries.

Status: Resolved

25. Miscellaneous comments

Severity: Informational

The following are suggestions to improve the overall code quality and readability.

- In `contracts/bl-market/src/liquidate.rs:241-246`, the validation is redundant as the `liquidate` function in line `105` already validated it in lines `68-77`. Consider removing the redundant validation.
- In `contracts/bl-market/src/execute.rs:351-360`, the `if` conditions can be refactored using the `if let Some(_) =` syntax to improve the code's readability and avoid executing `max_loan_to_value.is_some()` and `liquidation_threshold.is_some()`.
- In `contracts/bl-market/src/execute.rs:867`, the comment mentions that the function is used to repay native tokens. This is incorrect because the function handles all types of assets. Consider updating the comment.
- In `contracts/bl-market/src/execute.rs:913`, a match condition for `NotBorrowing` status is implemented. However, this will never be reached because if a user has no borrowings, the function will first error in line `900`. Consider returning an error if the match condition is entered.
- In `contracts/bl-market/src/execute.rs:1258-1262`, an event response is defined but commented out. Consider removing the commented code.
- In `contracts/bl-market/src/execute.rs:1274`, the comment incorrectly states the message's collateral pToken address. Consider removing the misleading comment.
- In `contracts/oracle/src/contract.rs:91`, the error is misleading because line `95` returns the same error. Consider using a different error to avoid confusion.
- In `contracts/oracle/src/contract.rs:105`, the `cancel_ownership` function is not descriptive. Consider implementing a more descriptive function name that indicates the proposed owner is being removed from storage and contract ownership is not being forfeited.
- In `contracts/liquidation-queue/src/state.rs:51`, a TODO comment is left in the code. Consider removing the TODO comment or implementing it.

Recommendation

We recommend applying the recommendations mentioned above.

Status: Resolved

26. Fixed price oracles can be set to zero

Severity: Informational

In `contracts/oracle/src/price_source.rs:90-92`, the `validate` function does not ensure that `OsmosisPriceSourceUnchecked::Fixed` is not configured with a zero price.

Recommendation

We recommend adding a validation to ensure the price being set is not zero.

Status: Resolved

27. Usage of magic numbers decreases maintainability

Severity: Informational

In `contracts/liquidation-queue/src/contract.rs:1289` and `1293`, hard-coded number literals without context or a description are used. Using such “magic numbers” goes against best practices as they reduce code readability and maintenance, and developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Resolved

Appendix

1. **Test case for "Liquidation queue contract receives fewer collaterals than expected when liquidating more than one asset"**

Please run the test case in `contracts/bl-market/src/contract.rs`.

```
#[test]
fn test_liquidate_double_asset_poc() {
	let available_liquidity_collateral = Uint128::from(1_000_000_000u128 /
2);
	let available_liquidity_native_debt = Uint128::from(1_000_000_000u128);

	let (mut deps, _env) = test_setup(&[
	coin(available_liquidity_collateral.into(), "collateral"),
	coin(available_liquidity_collateral.into(), "collateral2"),
	coin(available_liquidity_native_debt.into(), "native_debt"),
	]);

	let user_address = Addr::unchecked("user");
	let liquidator_address = Addr::unchecked("liquidator");

	let _collateral_price = Decimal::from_ratio(2_u128, 1_u128);
	let _native_debt_price = Decimal::from_ratio(15_u128, 10_u128);
	let user_collateral_balance = 2_000_000;
	let _user_debt = Uint128::from(3_000_000_u64); // ltv = 0.75

	let _second_debt_to_repay = Uint128::from(10_000_000_u64);
	let second_block_time = 16_000_000;

	let collateral_market_p_token_addr = Addr::unchecked("p_collateral");
	let collateral2_market_p_token_addr = Addr::unchecked("p_collateral2");

	let collateral_market = Market {
	ptoken_address: collateral_market_p_token_addr.clone(),
	max_loan_to_value: Decimal::from_ratio(5u128, 10u128),
	liquidation_threshold: Decimal::from_ratio(6u128, 10u128),
	debt_total_scaled: Uint128::new(800_000_000) * SCALING_FACTOR,
	liquidity_index: Decimal::one(),
	borrow_index: Decimal::one(),
	borrow_rate: Decimal::one(),
	liquidity_rate: Decimal::one(),
	reserve_factor: Decimal::one(),
	asset_type: AssetType::Native,
	indexes_last_updated: 0,
	..Default::default()
	};
```

```
        let collateral_market_initial =
        test_manual_init_market(deps.as_mut(), b"collateral",
&collateral_market);

        let collateral2_market = Market {
        ptoken_address: collateral2_market_p_token_addr.clone(),
        max_loan_to_value: Decimal::from_ratio(5u128, 10u128),
        liquidation_threshold: Decimal::from_ratio(6u128, 10u128),
        debt_total_scaled: Uint128::new(800_000_000) * SCALING_FACTOR,
        liquidity_index: Decimal::one(),
        borrow_index: Decimal::one(),
        borrow_rate: Decimal::one(),
        liquidity_rate: Decimal::one(),
        reserve_factor: Decimal::one(),
        asset_type: AssetType::Native,
        indexes_last_updated: 0,
        ..Default::default()
        };

        let collateral2_market_initial =
        test_manual_init_market(deps.as_mut(), b"collateral2",
&collateral2_market);

        let native_debt_market = Market {
        max_loan_to_value: Decimal::from_ratio(5u128, 10u128),
        debt_total_scaled: Uint128::new(100_000_000) * SCALING_FACTOR,
        liquidity_index: Decimal::one(),
        borrow_index: Decimal::one(),
        borrow_rate: Decimal::one(),
        liquidity_rate: Decimal::one(),
        reserve_factor: Decimal::one(),
        asset_type: AssetType::Native,
        indexes_last_updated: 0,
        liquidation_contract_address: Some(Addr::unchecked("liquidation")),
        ptoken_address: Addr::unchecked("p_native_debt"),
        ..Default::default()
        };

        let native_debt_market_initial =
        test_manual_init_market(deps.as_mut(), b"native_debt",
&native_debt_market);

        deps.querier.set_cw20_balances(
        collateral_market_p_token_addr.clone(),
        &[(
                user_address.clone(),
                Uint128::new(user_collateral_balance) * SCALING_FACTOR,
        )],
        );
```

```
        deps.querier.set_cw20_balances(
        collateral2_market_p_token_addr.clone(),
        &[(
                user_address.clone(),
                Uint128::new(user_collateral_balance) * SCALING_FACTOR,
        )],
        );

        deps.querier
        .with_liquidation_percent(&[(&"liquidation".to_string(),
&Decimal256::percent(95))]);

        deps.querier.with_oracle_price(&[(
        "collateral".to_string(),
        PriceResponse {
                denom: "collateral".to_string(),
                price: StdDecimal::percent(100),
        },
        )]);

        deps.querier.with_oracle_price(&[(
        "collateral2".to_string(),
        PriceResponse {
                denom: "collateral2".to_string(),
                price: StdDecimal::percent(100),
        },
        )]);

        deps.querier.with_oracle_price(&[(
        "native_debt".to_string(),
        PriceResponse {
                denom: "native_debt".to_string(),
                price: StdDecimal::percent(100),
        },
        )]);

        // Perform native liquidation
        {
        let mut user = User::default();
        set_bit(&mut user.collateral_assets,
collateral_market_initial.index).unwrap();
        set_bit(&mut user.collateral_assets,
collateral2_market_initial.index).unwrap();
        set_bit(&mut user.borrowed_assets,
native_debt_market_initial.index).unwrap();
        USERS
                .save(deps.as_mut().storage, &user_address, &user)
                .unwrap();

        let user_collateral_balance_scaled = Uint128::new(200) * SCALING_FACTOR;
```

```
        let expected_user_debt_scaled = Uint128::new(800) * SCALING_FACTOR;

        let debt_to_repay = Uint128::from(500u128);

        // Set the querier to return positive collateral balance
        deps.querier.set_cw20_balances(
                Addr::unchecked("p_collateral"),
                &[(user_address.clone(), user_collateral_balance_scaled.into())],
        );
        deps.querier.set_cw20_balances(
                Addr::unchecked("p_collateral2"),
                &[(user_address.clone(), user_collateral_balance_scaled.into())],
        );

        // set user to have positive debt amount in debt asset
        let debt = Debt {
                amount_scaled: expected_user_debt_scaled,
        };
        DEBTS
                .save(
                deps.as_mut().storage,
                (b"native_debt", &user_address),
                &debt,
                )
                .unwrap();

        let liquidate_msg = ExecuteMsg::Liquidate {
                debt_asset: Asset::Native {
                denom: "native_debt".to_string(),
                },
                borrower: user_address.to_string(),
        };

        let _collateral_market_before = MARKETS.load(&deps.storage,
b"collateral").unwrap();
        let _debt_market_before = MARKETS.load(&deps.storage,
b"native_debt").unwrap();

        let _block_time = second_block_time;
        let env = mock_env();
        //env.block.time. = block_time;
        let info = cosmwasm_std::testing::mock_info(
                liquidator_address.as_str(),
                &[coin(debt_to_repay.u128(), "native_debt")],
        );
        let res = execute(deps.as_mut(), env, info, liquidate_msg).unwrap();

        assert_eq![
                res.messages.len(),
                4 // this should be 5 : 2 TransferOnLiquidation, 2
```

```
ExecuteLiquidation, 1 RepayStableFromLiquidation
        ];

        assert_eq!(
                res.messages, // TransferOnLiquidation msg for p_collateral is
missing
                vec![
                SubMsg::new(CosmosMsg::Wasm(WasmMsg::Execute {
                        contract_addr: "p_collateral2".to_string(),
                        funds: vec![],
                        msg:
to_json_binary(&ptoken::msg::ExecuteMsg::TransferOnLiquidation {
                                sender: "user".to_string(),
                                recipient: "liquidation".to_string(),
                                amount: Uint128::from(189983081u128),
                        })
                        .unwrap(),
                })),
                SubMsg::new(CosmosMsg::Wasm(WasmMsg::Execute {
                        contract_addr: "liquidation".to_string(),
                        funds: vec![],
                        msg:
to_json_binary(&LiquidationQueueExecuteMsg::ExecuteLiquidation {
                                asset: Asset::Native {
                                denom: "collateral".to_string(),
                                },
                                collateral_amount:
Uint256::from(Uint128::from(9659u128)),
                                liquidator: "liquidator".to_string(),
                                repay_address: "cosmos2contract".to_string(),
                                fee_address: "protocol_rewards_collector".to_string()
                        })
                        .unwrap(),
                })),
                SubMsg::new(CosmosMsg::Wasm(WasmMsg::Execute {
                        contract_addr: "liquidation".to_string(),
                        funds: vec![],
                        msg:
to_json_binary(&LiquidationQueueExecuteMsg::ExecuteLiquidation {
                                asset: Asset::Native {
                                denom: "collateral2".to_string(),
                                },
                                collateral_amount:
Uint256::from(Uint128::from(9659u128)),
                                liquidator: "liquidator".to_string(),
                                repay_address: "cosmos2contract".to_string(),
                                fee_address: "protocol_rewards_collector".to_string()
                        })
                        .unwrap(),
                })),
```

```
            SubMsg::new(CosmosMsg::Wasm(WasmMsg::Execute {
                  contract_addr: "cosmos2contract".to_string(),
                  funds: vec![],
                  msg:
to_json_binary(&MarketExecuteMsg::RepayStableFromLiquidation {
                        borrower: "user".to_string(),
                        prev_balance: Uint256::from(1000000000u128),
                        debt_asset: Asset::Native {
                        denom: "native_debt".to_string()
                        },
                  })
                  .unwrap(),
            }))
            ]
      );
      }
}
```

2. Test case for "Market deposit limits may be surpassed"

Please run the test case in `contracts/bl-market/src/contract.rs`.

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

	////////////////// SETUP /////////////////

	let initial_liquidity = Uint128::from(10000000_u128);
	let (mut deps, mut env) = test_setup(&coins(initial_liquidity.into(),
"somecoin"));
	let reserve_factor = Decimal::from_ratio(1u128, 10u128);

	let deposit_coin_address = Addr::unchecked("ptoken");
	let mock_market = Market {
	ptoken_address: deposit_coin_address.clone(),
	liquidity_index: Decimal::from_ratio(11u128, 10u128),
	max_loan_to_value: Decimal::one(),
	borrow_index: Decimal::from_ratio(1u128, 1u128),
	borrow_rate: Decimal::from_ratio(10u128, 100u128),
	liquidity_rate: Decimal::from_ratio(10u128, 100u128),
	reserve_factor,
	debt_total_scaled: Uint128::new(10_000_000) * SCALING_FACTOR,
	deposit_cap_scaled: Uint128::new(20_000_000) * SCALING_FACTOR,
	indexes_last_updated: 10000000,
	..Default::default()
	};
	let market = test_manual_init_market(deps.as_mut(), b"somecoin",
&mock_market);

	// This is to mock the ptoken contract TokenInfo total_supply
	deps.querier.set_cw20_balances(
	deposit_coin_address.clone(),
	&[(Addr::unchecked("someuser"), Uint128::new(0))],
	);

	deps.querier.with_oracle_price(&[(
	"somecoin".to_string(),
	PriceResponse {
		denom: "somecoin".to_string(),
		price: StdDecimal::percent(100),
	},
	)]);

	// set symbol for ptoken contract
	deps.querier
	.set_cw20_symbol(deposit_coin_address.clone(), "somecoin".to_string());

	////////////////// FIRST DEPOSIT /////////////////
```

```
        let deposit_amount = 10_000_000;
        env.block.time = Timestamp::from_seconds(10000100);
        let info = mock_info("depositor", &coins(deposit_amount, "somecoin"));
        let msg = ExecuteMsg::DepositNative {
        denom: String::from("somecoin"),
        };
        let res = execute(deps.as_mut(), env.clone(), info, msg).unwrap();

        let expected_params = test_get_expected_indices_and_rates(
        &market,
        env.block.time.seconds(),
        initial_liquidity,
        Default::default(),
        );

        let expected_mint_amount = compute_scaled_amount(
        Uint128::from(deposit_amount),
        expected_params.liquidity_index,
        ScalingOperation::Truncate,
        )
        .unwrap();

        deps.querier.set_cw20_balances(
        deposit_coin_address.clone(),
        &[(Addr::unchecked("depositor"), expected_mint_amount)],
        );

        ////////////////// SECOND DEPOSIT /////////////////

        let deposit_amount = 10_000_000 + 2_000_000; //  over the deposit cap!
        env.block.time = Timestamp::from_seconds(10000900); // time advances ->
liquidity_index grows
        let info =
        cosmwasm_std::testing::mock_info("depositor", &coins(deposit_amount,
"somecoin"));
        let msg = ExecuteMsg::DepositNative {
        denom: String::from("somecoin"),
        };

        // execution does not revert
        execute(deps.as_mut(), env.clone(), info, msg).unwrap();
}
```