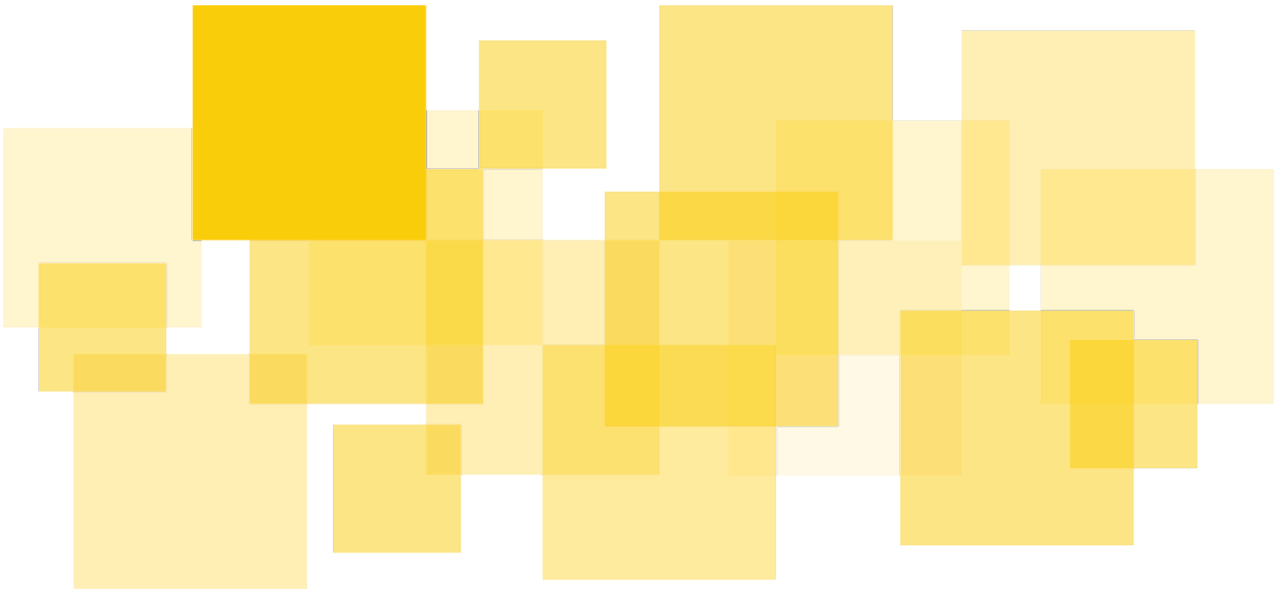
Security Audit Report

OctoLend Report Stellar

Delivered: March 20, 2026



runtime verification

Table of Contents

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks that otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis, and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carry substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims that impact a large quantity of funds.

Executive Summary

Untangled engaged Runtime Verification Inc. to conduct a security audit of the Collateral Vault and Lending Market contracts' code, encompassing the OctoLend platform. The objective was to review the platform’s business logic and implementation in Rust (Soroban) and identify any issues that could cause the system to malfunction or be exploited.

The audit was conducted over 5 calendar weeks (January 22, 2026, through February 27, 2026) and focused on analyzing the security of OctoLend's contracts' source code for its lending and borrowing operations, including a new delegation system that enables third parties to supply collateral on behalf of specific borrowers. Several other features are included in these contracts, which will be elaborated on later in this report.

The audit process involved a comprehensive review of the smart contract codebase, focusing on manual inspection and formal analysis techniques. Runtime Verification employed several techniques, including formal specification generation, invariant analysis, and testing, to rigorously analyze the system's behavior under various conditions. This included developing and analyzing key invariants to ensure the system's integrity across all state transitions.

Overall, code quality was strong, with clear structure, consistent patterns, and readable implementation. During the engagement, the Untangled team promptly responded to all reported issues and remediated them in a timely and effective manner. The team's fixes generally demonstrated a solid understanding of the underlying risks and led to meaningful improvements in the protocol’s safety.

The audit identified findings across a range of severities, from high to informative. These findings are grouped into the following categories:

Findings - issues with potential impact on protocol correctness, user funds, or core system behavior.

Informative Findings - general improvements, best-practice recommendations, and observations about clarity, maintainability, and future-proofing.

Runtime Verification performed targeted reviews of the implemented remediations and observed that the majority of fixes addressed the underlying issues as reported. As standard practice, we recommend that any substantial changes made after the audit period be reviewed as part of a follow-up assessment.

Overall, the Untangled team demonstrated strong responsiveness and a clear commitment to security. With the incorporated fixes, the audited codebase is materially more robust and better aligned with secure development best practices.

Scope

The scope of this audit was limited to the code contained in a private GitHub repository provided by the OctoLend team. The repository was cloned at commit `3f7a551be0aea05f4ac66889f481afa0b0b82ad3` on the `main` branch, which served as the initial revision under review. The elements within scope are divided into two core components:

- `./contracts/lending-market` : Contains the implementation of the lending market contract, including all borrowing and lending logic, collateral management, an adaptive interest rate model (IRM), flash loan functionality, liquidation mechanics, oracle integration, and administrative configuration endpoints. Approximately 2,389 lines of Rust across 13 source files.
- `./contracts/collateral-vault` : Implements a tokenized vault following the SEP-0056 pattern (Soroban's approximation of Solidity's ERC-4626), with additional delegation capabilities that allow vault depositors to allocate collateral to lending markets on behalf of borrowers. Includes deposit/withdrawal operations, allocation and deallocation flows, delegation fee management, and cross-contract liquidation handling. Approximately 930 lines of Rust across 8 source files.

The `./contracts/clients` directory, containing client interface stubs used for cross-contract invocations, was reviewed for context but was not a primary audit target. Frontend logic, deployment infrastructure, off-chain tooling, and third-party integrations (including oracle data feed providers) are explicitly excluded from this engagement.

As findings were identified during the review, the client submitted several commits addressing reported issues. These remediation commits were also reviewed to verify that the identified issues were appropriately resolved prior to report finalization. The latest reviewed commit is `[6b6d473]` .

Methodology

Runtime Verification followed a structured methodology to evaluate the correctness and security of the OctoLend smart contracts within the agreed engagement timeline. Although manual review cannot guarantee the discovery of all vulnerabilities, our approach was designed to maximize coverage and identify the most impactful risks.

Design Review and Invariant Identification

The engagement began with a high-level design review of the OctoLend protocol, focusing on the lifecycle of its Lending Market operations and the management of assets through delegation in Collateral Vaults.

To contextualize the intended behavior, we also utilized the documentation supplied by the client, which informed our understanding of expected invariants and economic properties relevant to the involved systems.

This phase produced a set of functional and security invariants that guided the detailed review.

Manual Code Review

After establishing the protocol model, we performed a line-by-line review of the Soroban Rust codebase. This included examining:

- Correctness of collateral and borrow accounting and management within the Lending Markets;
- Correctness of the delegation and allocation systems in the Collateral Vaults;
- Authorization and permission boundaries;
- Token handling and mathematical logic;
- Edge cases, boundary conditions, and potential state inconsistencies.

Execution paths and state transitions were evaluated against the identified invariants to ensure alignment between the intended design and implemented behavior.

Tool-Assisted Analysis

To supplement manual review, we incorporated targeted automated analysis:

- **RV Audit Assistant** for AI-driven precondition analysis and vulnerability detection;
- **Almanax** for AI-assisted bug pattern detection and anomaly surfacing;
- **Komet** for property-based fuzzing on selected components, with a focus on the protocol's most critical execution paths.

These tools enhanced coverage but were used in a supporting role, with manual reasoning driving the core assessment.

Additionally, we used `cargo audit`, `cargo outdated`, `cargo clippy`, and `cargo tarpaulin` to look for potential issues, with no relevant information discovered other than particularities of the libraries used by the Soroban infrastructure.

Finally, we conducted rounds of internal discussions with security experts over the code and platform design to verify possible exploitation vectors and identify improvements for the analyzed contracts.

Additionally, given the nascent Soroban development and auditing community, we reviewed this list of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to the smart contracts and scripts; if they apply, we checked whether the code is vulnerable to them.

Fuzz Testing with Komet

To complement manual code review, we employed Komet — a fuzzing framework for Soroban smart contracts — to stress-test the protocol's most critical execution paths. Our approach used the project's existing Rust unit tests as a foundation, translating them into Komet's contract-based harness and parameterizing their fixed inputs, allowing the fuzzer to explore a broad range of values systematically.

Two areas were prioritized based on their criticality to protocol correctness and user funds.

The first is the borrow and repay flow in the lending market. This flow is the core of the protocol's economic model: it governs debt token minting, interest accrual via the IRM, collateral validation, and liquidity accounting. We parameterized the supply amount, borrow amount, and repay amount, with assume! guards derived from the contract's own invariants — such as borrow capacity and available liquidity — to keep inputs in the valid state space. The fuzzer then verified that post-borrow and post-repay accounting (debt shares, debt assets, token balances) remained consistent across all explored inputs.

The second is the delegation, allocation, and deallocation flow in the collateral vault. This flow is critical because it governs how collateral is delegated from depositors to borrowers and allocated to lending markets. Errors here could allow undercollateralized borrowing or incorrect capacity accounting. We parameterized the delegate, allocate, and deallocate amounts and verified that the vault's internal state — delegated balances, remaining capacity, and allocated amounts — remained self-consistent throughout.

By grounding our fuzz tests in the existing test suite, we ensured coverage of the intended happy paths while extending confidence to the broader input space. No invariant violations were discovered during fuzzing, which increases our trust in the correctness of these flows under varied conditions.

These tests will be contributed to the protocol's repository as a Pull Request once it is made public. They are fully self-contained and executable with a single command: `komet test`.

Platform Features and Description

OctoLend is a lending and borrowing system on the Stellar blockchain. It has two main parts: lending markets and collateral vaults. In a lending market, people can deposit an asset (e.g., a stablecoin) to earn interest, and others can borrow that asset by posting collateral. The collateral vault is optional: it lets third parties (“delegators”) deposit collateral and assign it to borrowers, so borrowers can get more borrowing capacity without locking more of their own tokens. The protocol uses an oracle for collateral prices, and liquidations protect lenders when a borrower’s collateral is no longer enough to cover their debt.

The system is built as smart contracts: one contract per lending market (which also contains the interest rate logic), and one contract per collateral vault. Admins configure each market (e.g., liquidation rules, fees) and can link a vault to a market so delegation is allowed. There is no separate “IRM contract”; the interest rate logic lives inside each lending market contract.

Collateral Vaults

A collateral vault is a place where delegators deposit one type of token (the collateral asset). They do not borrow themselves; instead, they delegate some of that balance to specific borrowers. Those borrowers can then allocate that delegated amount to a lending market, so the market counts it as extra collateral when computing how much the borrower can borrow. The vault tracks how much is delegated to each borrower and how much each borrower has allocated to each market, and it can charge delegation fees. Borrowers can allocate and deallocate over time, and the vault checks that they still meet minimum collateral and allocation limits.
The vault does not mint, burn, or move tokens on behalf of the lending market. The lending market only reads from the vault (e.g. “how much is allocated to this borrower for this market?”) to decide borrowing capacity. Deposit, withdraw, delegate, allocate, and fee logic are all in the vault; the market just uses the numbers it reads. One vault can serve multiple lending markets because allocations are stored per (borrower, market) pair.

Breaking down the available endpoints of Collateral Vaults into functionality groups, we have:

1. Constructor

- `__constructor (asset, decimals_offset, delegator, admin)` : One-time initialization. It sets the vault’s underlying asset, decimals offset, delegator address, and admin.

2. Delegation (delegator-only)

- `delegate (caller, to, amount)` : Delegator adds amount of collateral to borrower to. Caller must be the delegator.
- `undelegate (caller, to, amount)` : Delegator reduces by amount the collateral delegated to borrower to. Caller must be the delegator.

3. Delegation views (read-only)

- `get_delegated (to)` : Returns total collateral currently delegated to address to.
- `get_total_delegated()` : Returns total collateral delegated to all borrowers.
- `remaining_capacity()` : Returns how much of the vault’s balance is not delegated (available to delegate).

4. Receiver config (admin-only)

- `set_receiver_config (admin, receiver, max_allocation_rate, min_collateral, delegation_fee_bps)` : Admin sets a borrower’s config: max allocation rate, min collateral, delegation fee (bps). Only allowed when that receiver has zero delegated balance.

5. Allocation (borrower-only)

- `allocate (borrower, market, amount)` : Borrower assigns amount of their delegated balance to market. Caller must be borrower.
- `deallocate (borrower, market, amount)` : Borrower removes amount of allocation from market back to their delegated balance. Caller must be borrower.

6. Allocation views (read-only)

- `get_allocated (borrower, market)` : Returns allocation info for borrower at market: amount, timestamp, accrued fee.
- `get_receiver_config (receiver)` : Returns receiver config for receiver: max_allocation_rate, min_collateral, delegation_fee_bps.

7. Liquidation and fees

- `liquidate_allocation (liquidator, borrower, market, timestamp)` : Liquidator claims their allotted collateral for a past liquidation at market for this (borrower, timestamp). Caller must be liquidator.
- `pay_delegation_fee (borrower, market, amount)` : Borrower pays amount of accrued delegation fee to the vault. Caller must be borrower.

8. Vault token (FungibleVault / FungibleToken)

- `deposit(assets, receiver, from, operator)` : Operator deposits assets of underlying asset on behalf of from; receiver gets vault shares. Returns shares minted.
- `withdraw(assets, receiver, owner, operator)` : Operator withdraws assets of underlying for owner; receiver gets the assets. Withdrawable amount is capped by non-delegated balance. Returns shares burned.
- `mint(shares, receiver, from, operator)` : Operator mints shares for receiver by depositing underlying from from. Returns assets deposited.
- `redeem(shares, receiver, owner, operator)` : Operator redeems shares of owner; receiver gets underlying. Redeemable shares capped by non-delegated share value. Returns assets withdrawn.
- `query_asset()` : Returns the underlying asset address.
- `total_assets()` : Returns total underlying assets held by the vault.
- `convert_to_shares(assets)` : Converts an asset amount to vault shares (preview).
- `convert_to_assets(shares)` : Converts vault shares to asset amount (preview).
- `max_deposit(receiver)` : Max assets that can be deposited for receiver.
- `preview_deposit(assets)` : Shares that would be minted for depositing assets.
- `max_mint(receiver)` : Max shares that can be minted for receiver.
- `preview_mint(shares)` : Assets required to mint shares.
- `max_withdraw(owner)` : Max assets withdrawable by owner (limited by non-delegated balance).
- `preview_withdraw(assets)` : Shares that would be burned to withdraw assets.
- `max_redeem(owner)` : Max shares redeemable by owner (limited by non-delegated value).
- `preview_redeem(shares)` : Assets that would be received for redeeming shares.
- `decimals()` : Returns vault token decimals (overrides base).

The vault also implements the standard FungibleToken interface (e.g. `balance` , `transfer` , `allowance` , `approve`) from `stellar_tokens` ; those are the usual token read/write endpoints on top of the ones above.

Lending Markets

A lending market is a single pool for one borrowed asset (e.g. a stablecoin) and one collateral asset. Lenders deposit the borrowed asset and receive a share token that grows in value as interest accrues; they can withdraw later by redeeming shares. Borrowers first supply collateral (the collateral asset), then borrow the borrowed asset up to a limit based on their collateral value and a liquidation

threshold (e.g. 80% loan-to-value). They repay with the borrowed asset; repayment burns “debt shares” and reduces what they owe. An oracle provides the collateral price so the market can value collateral and enforce limits.

The market also handles liquidations: if a borrower’s debt value exceeds the allowed fraction of their collateral value (e.g. over 80%), anyone can repay part or all of their debt in exchange for collateral, plus a small liquidation bonus. The market can be linked to a collateral vault; when enabled, a borrower’s borrowing capacity includes both their own collateral and any collateral delegated to them and allocated to that market. Fees (e.g. on interest) can be sent to a configurable beneficiary. The market stores one collateral vault address per market and does not perform vault operations itself—it only reads from the vault.

Breaking down the available endpoints of Lending Markets into functionality groups, we have:

1. Constructor

- `__constructor(asset, decimals_offset, delegator, admin)` : One-time initialization. Sets the vault’s underlying asset, decimals offset, delegator address, and admin.

2. Admin Configurations - IRM & Oracle

- `set_irm_config(caller, config)` : Admin sets IRM config once (kink, rates, etc.). Caller must be admin.
- `set_oracle(caller, oracle)` : Admin sets oracle (immediate if first time; otherwise starts 48h timelock).
- `execute_oracle_update(caller)` : Admin applies pending oracle after timelock.
- `cancel_oracle_update(caller)` : Admin cancels pending oracle.
- `get_pending_oracle_update()` : Returns pending oracle address and execution timestamp (read-only).

3. Admin Configurations - Vault, Delegation, Market Config

- `set_collateral_vault(caller, vault)` : Admin sets linked collateral vault.
- `set_delegate_enabled(caller, enabled)` : Admin turns delegation on/off.
- `set_liquidation_threshold(caller, threshold)` : Admin sets liquidation LTV (e.g. 8000 = 80%). Only before market config is set.
- `set_liquidation_bonus_bps(caller, bonus_bps)` : Admin sets liquidation bonus in bps (e.g. 500 = 5%). Only before market config is set.
- `set_fee(caller, fee_bps)` : Admin sets protocol fee on interest (bps).
- `set_market_config(caller, liquidation_threshold, liquidation_bonus_bps, fee_bps)` : Admin sets liquidation threshold, bonus, and fee in one call; only callable once.
- `set_fee_beneficiary(caller, beneficiary)` : Admin sets fee recipient.

4. Admin Configurations - Protocol Pause

- `pause_contract(caller)` : Admin pauses all state-changing user operations.
- `unpause_contract(caller)` : Admin unpauses.

5. Read-only - Config & State

- `get_collateral_vault()` : Returns linked collateral vault address.
- `get_collateral_asset()` : Returns collateral asset address.
- `get_oracle()` : Returns oracle address.
- `get_utilization_rate()` : Returns current utilization (debt / total assets).
- `get_total_debt_assets()` : Total debt in underlying asset units (shares × price).
- `get_total_debt_shares()` : Total debt token supply (outstanding debt shares).
- `get_last_debt_assets()` : Snapshot of total debt in asset units at last accrual/borrow/repay.
- `get_last_total_assets()` : Snapshot of total pool size (debt + liquidity) at last update.

- `get_irm_price()` : Current debt token price (asset per share, 7 decimals).
- `get_irm_rate()` : Current interest rate (APR-style, from IRM).

6. Read-only - Borrower Data

- `get_collateral_amount(borrower)` : Borrower's collateral balance in the market.
- `get_borrowed_amount(borrower)` : Borrower's debt in debt shares.
- `get_borrowed_value(borrower)` : Borrower's debt in underlying asset units.
- `get_borrow_capacity(borrower)` : Max additional borrow in asset units (from LTV and collateral).
- `is_healthy(borrower)` : True if LTV ≤ liquidation threshold.

7. Borrower / Lender Operations (require auth, revert when paused)

- `borrow (borrower, amount)` : Borrower borrows amount of underlying asset; receives tokens, debt shares increase.
- `repay (borrower, amount)` : Borrower repays amount of underlying; debt shares decrease.
- `supply_collateral (borrower, amount)` : Borrower adds amount of collateral asset to the market.
- `withdraw_collateral (borrower, amount)` : Borrower withdraws amount of collateral if still above LTV limit.

8. Liquidation

- `available_liquidation(borrower)` : Max debt value (in asset terms) that can be liquidated for this borrower (0 if healthy).
- `liquidate (liquidator, borrower, repaid_shares, seized_collateral)` : Liquidator repays debt (in shares) and receives collateral (with bonus). Caller must be liquidator.
- `get_pending_seize(liquidator, borrower, timestamp)` : Amount of collateral pending for this liquidator for a given liquidation (timestamp).

9. Flash Loan

- `flash_loan(caller, receiver, token, amount, data)` : Caller initiates flash loan: market sends amount of token (must be market asset) to receiver, calls receiver.on_flash_loan(...), then pulls amount back; reverts if not repaid.

10. Vault Token (FungibleVault / FungibleToken)

- `deposit(assets, receiver, from, operator)` : Operator deposits assets of underlying for from; receiver gets market (lender) shares. Accrues interest and fee first.
- `withdraw(assets, receiver, owner, operator)` : Operator withdraws assets of underlying for owner; receiver gets assets. Accrues interest and fee first.
- `mint(shares, receiver, from, operator)` : Operator mints shares for receiver by depositing underlying from from. Accrues interest and fee first.
- `redeem(shares, receiver, owner, operator)` : Operator redeems shares of owner; receiver gets underlying. Accrues interest and fee first.
- `query_asset()` : Returns underlying (borrowed) asset address.
- `total_assets()` : Total pool size (debt + available liquidity).
- `convert_to_shares(assets)` : Shares that would be minted for depositing assets.
- `convert_to_assets(shares)` : Underlying that would be received for redeeming shares.
- `max_deposit(receiver)` : Max assets that can be deposited for receiver.
- `preview_deposit (assets)` : Shares that would be minted for depositing assets.
- `max_mint(receiver)` : Max shares that can be minted for receiver.
- `preview_mint(shares)` : Assets required to mint shares.
- `max_withdraw(owner)` : Max assets withdrawable by owner.

- `preview_withdraw(assets)` : Shares that would be burned to withdraw assets.
- `max_redeem(owner)` : Max shares redeemable by owner.
- `preview_redeem(shares)` : Assets that would be received for redeeming shares.
- `decimals()` : Market (lender) share token decimals (overrides base).

The contract also exposes the usual FungibleToken interface from stellar_tokens (e.g. `balance` , `transfer` , `allowance` , `approve`), so lender shares behave as a standard token on top of the endpoints above.

Interest Rate Model (Inside the Lending Market)

The interest rate model (IRM) is logic embedded in each lending market contract, not a separate contract. It decides how interest accrues on borrowed amounts. Debt is represented as “debt shares”; the price of one debt share in terms of the borrowed asset increases over time as interest is applied. When someone borrows, they receive debt shares; when they repay, they burn shares. The IRM updates a stored “debt token price” and a snapshot of total debt in asset terms (last debt assets) whenever interest is compounded (e.g., on borrow, repay, or when the rate is read). So “total debt shares” is the number of shares outstanding; “last debt assets” is the total borrowed amount in underlying asset units at the last update.
The rate itself depends on utilization: the fraction of the pool that is borrowed (debt assets ÷ total assets). The IRM uses a “kink” style curve with a target utilization and parameters that push the rate up when utilization is high and down when it is low, so the protocol attracts liquidity when it’s scarce and encourages borrowing when it’s abundant. All of this uses fixed-point math (no floats), so behavior is deterministic and safe on-chain. There is one IRM configuration per lending market, stored in that market’s state.

The `clients` module

Included in the engagement's scope, the `clients` module contains lightweight Soroban client interface definitions used for cross-contract communication between the protocol's components. Each file declares a trait with the necessary implementations to auto-generate typed client stubs at compile time. These stubs allow one contract to invoke functions on another in a type-safe manner. The crate defines three such interfaces: `VaultClient` for querying allocation state and receiver configuration from the Collateral Vault, `MarketClient` for reading lending market state such as collateral amounts, debt accounting, and unallocation validation, and `InterestRateModelClient` for triggering interest compounding and retrieving rate and pricing data from the IRM.

This crate does not contain any business logic of its own; it serves purely as a shared dependency that both the Lending Market and Collateral Vault contracts import so that each can invoke the other's endpoints without duplicating type definitions. It also re-exports the Allocation and ReceiverConfig struct types used in cross-contract call parameters and return values

OctoLend's Lifecycle

Phase 1 - Deployment and Configuration

The protocol administrator deploys two contracts: a **Lending Market** and a **Collateral Vault**. Each Lending Market is instantiated with a specific lending asset (e.g., USDC) and a collateral asset (e.g., XLM). During initialization, the admin configures the market parameters (liquidation threshold, liquidation bonus, fee basis points) and sets up the Interest Rate Model (IRM) configuration, oracle address, and fee beneficiary. On the Collateral Vault side, the admin sets the underlying collateral asset, assigns a delegator role, whitelists lending markets that the vault is allowed to interact with, and configures per-receiver parameters such as the maximum allocation ratio, minimum collateral requirement, and delegation fee rate. Finally, the admin links the Lending Market to the Collateral Vault by calling `set_collateral_vault` , and optionally enables delegated collateral via `set_delegate_enabled` .

Phase 2 - Supplying Liquidity

Lenders deposit the lending asset (e.g., USDC) into the Lending Market through the `deposit` endpoint, which follows the SEP-0056 tokenized vault pattern. The market compounds accrued interest, accrues protocol fees, converts the deposited assets into proportional vault shares, and mints those shares to the lender. These shares represent the lender's claim on the pool's underlying

assets plus any interest earned over time. Lenders can exit at any point by calling `withdraw` or `redeem`, burning their shares in exchange for the underlying assets.

Phase 3 - Supplying Collateral and Borrowing

A borrower who wants to take a loan first calls `supply_collateral`, transferring the collateral asset (e.g., XLM) into the Lending Market contract. This collateral is recorded per-borrower and determines their borrowing capacity via the Loan-to-Value (LTV) ratio enforced by the liquidation threshold. Once collateral is posted, the borrower calls `borrow`, specifying how much of the lending asset they want to receive. The market checks that the loan keeps the position healthy, mints debt shares representing the borrower's obligation, transfers the requested assets to the borrower, and updates the IRM's utilization-dependent interest rate.

Phase 4 - Delegated Collateral (Optional Path)

Alternatively, a borrower can source collateral through the Collateral Vault instead of posting it directly. In this flow, the vault's delegator first calls `delegate` to assign a credit line to the borrower from the vault's pooled collateral. The borrower then calls `allocate` on the Collateral Vault, directing a portion of their delegated amount to a specific whitelisted lending market. This allocated amount counts as collateral in the borrower's health check on that market, enabling them to borrow without depositing their own collateral tokens. In exchange, the borrower accrues a delegation fee proportional to the allocated amount and duration. The borrower can later `deallocate` to reduce their allocation and must periodically call `pay_delegation_fee` to settle accrued fees.

Phase 5 - Interest Accrual and Rate Adaptation

As time passes, the IRM continuously adapts the interest rate based on pool utilization. Each time a state-changing operation occurs (deposit, borrow, repay, withdraw), `compound_interest` is called, which computes the accrued interest since the last update, increases the debt token price (making each debt share worth more of the underlying asset), and adjusts the IRM's internal rate using a PI-controller mechanism. When utilization exceeds a target threshold, the rate accelerates upward; when utilization is below the target, the rate decreases. The protocol also takes a percentage of the accrued interest as fees, minting additional vault shares to the designated fee beneficiary.

Phase 6 - Repayment

When the borrower is ready to close or reduce their debt, they call `repay` with the amount of the lending asset they wish to return. The market calculates how many debt shares the repayment covers (based on the current debt token price, which has grown with interest), burns those shares, and transfers the lending asset from the borrower back into the pool. If the borrower repays their full debt, all their debt shares are burned. With reduced or eliminated debt, the borrower can then call `withdraw_collateral` to reclaim some or all of their posted collateral, subject to the health check ensuring any remaining debt is still adequately collateralized.

Phase 7 - Liquidation

If a borrower's position becomes unhealthy, any third party can call `liquidate`. The liquidator repays a portion of the borrower's debt (burning debt shares) and, in exchange, seizes collateral at a discount determined by the liquidation bonus. For directly-posted collateral, the seized tokens are transferred to the liquidator immediately. For delegated collateral, the market records a `PendingSeize` entry, and the liquidator must then call `liquidate_allocation` on the Collateral Vault in a second step, which verifies the pending seize, reduces the borrower's allocation, accrues any outstanding delegation fees, forfeits the borrower's unpaid fees, and transfers the seized collateral tokens from the vault to the liquidator.

Phase 8 - Flash Loans

At any point, a user can execute a flash loan via the `flash_loan` endpoint, borrowing any amount of the pool's available assets without collateral, as long as the full amount is returned within the same transaction. The market transfers the requested tokens to a receiver contract, invokes a callback on that receiver to perform arbitrary logic, and then pulls the tokens back. If the balance is not fully restored, the transaction reverts.

Invariants

During the audit, invariants were defined and used to guide part of our search for possible issues with OctoLend's smart contracts. Using the previously explored specifications, the client's documentation, the intended business logic, and references collected during the audit, we identified the following invariants for the protocol's smart contracts:

High-level invariants

This review adopts a trust model in which the protocol administrator is considered a fully trusted party. This assumption extends beyond the conventional notion of honest behavior to encompass operational availability: the admin is assumed to be both willing and able to configure parameters correctly, monitor protocol health, and deploy contract updates or remediation measures as needed.

Lending market

A. Pool and accounting

1. Pool conservation (borrowed asset)
   At all times, `total_assets = total_debt_assets + available_liquidity`. The contract's balance of the borrowed asset equals `available_liquidity`; the rest of `total_assets` is the debt value (`shares x price`).
2. Debt consistency
   `total_debt_assets = total_debt_shares x debt_token_price / 10^7` (up to rounding). The snapshot `last_debt_assets` is updated on every accrual, borrow, and repay and is used for utilization/IRM. It should match the derived total debt in asset terms at those update points.
3. Share-asset monotonicity
   Lender share value does not decrease from rounding:
   `preview_redeem(redeem_amount) ≤ preview_redeem(redeem_amount + 1)` and
   `preview_deposit(assets) / preview_withdraw(assets)` are consistent with `convert_to_shares / convert_to_assets` and `total_assets`.
4. Utilization bounds
   `0 ≤ utilization_rate ≤ 10_000` (0%-100% in bps).
   `utilization = total_debt_assets / total_assets when total_assets > 0`; 0 when `total_assets = 0`.

B. Borrowing and collateral

1. Borrow limit
   No borrow is allowed unless `amount ≤ get_borrow_capacity(borrower)` and `amount ≤ available_liquidity`. Borrow capacity is derived from `(collateral + delegated allocation)` value and liquidation threshold.
2. Collateralization
   After every `withdraw_collateral`, for that borrower
   `borrowed_value x 10_000 ≤ principal_value x liquidation_threshold_bps` (principal includes direct collateral + delegated allocation from the vault).
3. Liquidation only when unhealthy
   liquidate only runs when `!is_healthy(borrower)` (i.e. LTV > liquidation threshold). `available_liquidation(borrower)` is 0 when healthy.
4. Liquidation bounds
   `repaid_shares ≤ get_borrowed(borrower)` and `seized_collateral ≤ user_collateral + delegated` for that borrower.
   `pending_seize` is set so the same liquidation slice is not claimed twice.

C. Config and access

1. Immutable / one-time config
   `set_irm_config` and `set_market_config` are callable only once (guarded by `is_irm_config_set` / `is_market_config_set`). `asset` and `collateral` are set only in the constructor.
2. Pause
   When the contract is paused, every state-changing user entrypoint (borrow, repay, supply/withdraw collateral, deposit/withdraw/mint/redeem, liquidate, flash_loan) reverts; only admin config and getters can be used.
3. Privileged calls
   All `set_*` config, pause_contract, unpause_contract, and oracle timelock (execute/cancel) functions require caller to be admin and require_auth(caller).
4. Actor auth
   `borrow / repay / supply_collateral / withdraw_collateral` require auth of the borrower; `liquidate` requires auth of the liquidator; `deposit / withdraw / mint / redeem` require auth of the operator.

Collateral vault

A. Delegation and allocation

1. Delegation capacity
   `total_delegated ≤ vault_balance` (underlying asset). `remaining_capacity = vault_balance - total_delegated`. `delegate` fails if it would make `total_delegated` exceed the vault balance.
2. Allocation vs delegation
   For every (borrower, market), `allocated(borrower, market).amount ≤ get_delegated(borrower)`. `total_allocated` is the sum of all allocation amounts and satisfies `total_allocated ≤ total_delegated`.
3. Withdrawable cap
   `max_withdraw(owner)` and `max_redeem(owner)` are limited by `total_assets - total_delegated` (and share conversion), so users cannot withdraw more than the non-delegated portion of the pool.
4. Receiver config lock
   `set_receiver_config` for a receiver is disallowed when `get_delegated(receiver) > 0`, so config cannot change while they have an active delegation.

B. Allocation rules

1. Allocate / deallocate
   `allocate` only if `delegated(borrower) ≥ amount` and receiver config (min_collateral, max_allocation_rate) is satisfied for that borrower at the target market. deallocate only if `allocated(borrower, market) ≥ amount`.
2. Liquidation of allocation
   `liquidate_allocation` only transfers collateral up to `min(pending_seize, allocation.amount)` and sets `IsLiquidated(liquidator, borrower, timestamp)` so the same liquidation cannot be claimed again.
3. Fee accounting
   `pay_delegation_fee` and `forfeit_fees` only reduce `allocation.accrued_fee` by at most the current accrued amount. `TotalFeesAccrued` and fee reductions are consistent, so the total accrued fees never go negative.

C. Access and roles

1. Delegation auth
   `delegate` and `undelegate` require `only_delegator(caller)` and `require_auth(caller)`. allocate and deallocate require `require_auth(borrower)` with `caller = borrower`.
2. Admin and config
   `set_receiver_config` requires `only_admin(admin)` and `require_auth(admin)`. Admin is set once in the constructor.

3. Vault token behavior
   `deposit / withdraw / mint / redeem` use `convert_to_shares / convert_to_assets` and `respect max_withdraw / max_redeem` so that withdrawals never exceed the non-delegated balance and share/asset math is consistent.

D. Cross-contract (when delegation is enabled)

1. Borrow capacity
   In the lending market, `get_borrow_capacity(borrower)` uses `get_collateral(borrower) + get_delegated(borrower)` (delegated = vault's `get_allocated(borrower, market).amount` for this market). So the market never assumes more delegated collateral than the vault reports for that (borrower, market).
2. Withdraw collateral
   When the market's `withdraw_collateral` checks delegation constraints, it uses the vault's `get_receiver_config` and `get_allocated`; the market does not allow a withdrawal that would violate the vault's `allocation/min_collateral/max_allocation_rate` rules.

Findings

This section contains all issues identified during the audit that could lead to unintended behavior, security vulnerabilities, or failure to enforce the protocol's intended logic. Each issue is documented with a description, potential impact, and recommended remediation steps.

[A1] Malicious Delegated Users Can Fully Withdraw the Collateral Vault Balance

Severity: High | Difficulty: High | Recommended Action: Fix Design | Addressed by client

Description

The Collateral Vault's `allocate` logic did not reject non-positive `amount`s. The `amount` parameter defines how much of the delegated collateral can be used by the borrower in a lending market, and, for a borrower with a positive delegated balance, calling allocate with a negative amount may cause an increase in its delegated balance. This happens as the delegated balance is updated by subtracting any amount being allocated by the borrower (`delegated - amount`). When `amount` is negative, that operation becomes `delegated + |amount|` , so the borrower's delegated balance increased without the delegator's consent.

A complicating factor is that the Collateral Vault allows its users to supply the addresses of the Lending Markets for its internal operations. The only prerequisite for operations in the Collateral Vault involving Lending Markets to be executed is that the address supplied by the user implements the Lending Market interface.

A malicious user can implement a custom Lending Market that, for instance, exposes a `get_collateral_amount` endpoint reporting that they hold an arbitrary amount of collateral in that market. This can be used to bypass the minimum-deposited collateral validation in the `allocate` function, which would prevent the malicious user from allocating arbitrary amounts to that specific Lending Market. Similarly, when a liquidator calls `liquidate_allocation` , the vault does not check that the provided market address is an official, trusted lending market. It only uses the Lending Market interface: it calls that address's `get_pending_seize` and trusts the return value. A malicious contract can report an arbitrary pending seize amount (e.g., the full vault balance) even though no real liquidation occurred and no real debt exists. The vault has no way to tell a legitimate market from a fake one.

The combination of the previously discussed issues results in the total drainage of the collateral vault: all depositors' collateral can be taken by the attacker or the liquidator, with no real lending or liquidation having occurred. The attack relies only on (1) the vault accepting negative allocation amounts and (2) the vault trusting any address that implements the Lending Market interface for get_pending_seize.

Scenario

The following is the list of theoretical steps exemplifying this exploit:

```
1 - Malicious user Alice convinces the delegator to allocate some collateral to them (e.g., proposes to make a tool for the
delegators or anything of the delegator's interest);
2 - Delegator allocates 10 of the 1000 tokens held in the Collateral Vault to Alice;
3 - Alice deploys a fake Lending Market M (same interface but custom, malicious logic);
4 - Alice allocates -990 collateral to any Lending Market that is not M. Delegation to Alice becomes 10 - (-990) = 1000;
5 - Alice allocates 1000 tokens to M;
6 - Alice (or someone on their behalf) calls liquidate_allocation targeting Alice as the borrower, M as the market, and an
arbitrary timestamp;
7 - M, a custom malicious contract with the Lending Market interface, returns 1000 as the pending_seize, causing Alice to withdraw
the full balance of the collateral vault.
```

Step 4 requires Alice to have enough collateral on a legitimate Lending Market to pass the collateral rate check in the Collateral Vault, but this can be bypassed by using a malicious Lending Market (with a custom return for `get_collateral_amount`) instead.

Recommendation

Fixing the negative amount check prevents the delegation inflation step, and restricting which addresses may be used as the "market" in liquidate_allocation (e.g., to a whitelist of known lending markets) prevents the fake collateral amount and pending seize from being used to drain the vault.

Status

This issue has been resolved in commit `d1e92e1`. The fix adds validation in `allocate` to reject non-positive amounts, preventing delegation inflation via negative allocations, and introduces a whitelist of approved lending markets. The whitelist is enforced during allocation so that unknown market addresses are rejected, removing the ability to use malicious contracts to fake collateral or pending seize values and thereby eliminating the vault-drain vector.

[A2] Liquidations in the Collateral Vault Don't Update `TotalAllocated`

Severity: Low | Difficulty: Medium | Recommended Action: Fix Code | Addressed by client

Description

The collateral vault keeps a global counter `TotalAllocated` that should equal the sum of all per-borrower, per-market allocation amounts. When a borrower calls allocate, the vault increases both the borrower's allocation and `TotalAllocated`. When they call deallocate, it decreases both. However, when a liquidator calls `liquidate_allocation` to seize collateral from a borrower's allocation, the vault reduces the borrower's allocation and transfers the seized collateral, but it does not decrease `TotalAllocated` by the seized amount. As a result, `TotalAllocated` becomes stale and overstated after any liquidation.

This breaks the intended invariant that `TotalAllocated` equals the sum of all allocation amounts. After liquidations, `TotalAllocated` can be arbitrarily higher than the actual sum of allocations. While the current code does not read `TotalAllocated` for business logic (withdraw limits use `TotalDelegated` instead), this inconsistency can mislead off-chain monitoring, break future code that relies on `TotalAllocated`, and make it harder to comprehend the vault's state.

Recommendation

The fix is to decrement `TotalAllocated` by the seized amount in `liquidate_allocation`, matching the pattern used in allocate and deallocate.

Status

This finding has been addressed in commit `d1e92e1` by following the above recommendation.

[A3] There Are No Protections Against Stale Oracle Prices In Lending Markets

Severity: Medium | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The lending market defines a `MAX_STALENESS` constant (3600 seconds, one hour) to limit how old an oracle price can be before it is considered stale. The `get_price` function includes a check that compares the current ledger timestamp to the price data timestamp from the oracle. If the difference exceeds `MAX_STALENESS`, the code would panic and reject the stale price. However, this check is commented out, so the contract accepts prices regardless of age.

This allows stale prices to be used for critical operations such as calculating collateral values, determining borrow capacity, and executing liquidations. If the oracle stops updating or experiences delays, the market can continue using outdated prices, leading to incorrect valuations, incorrect liquidation decisions, and potential economic losses for users.

Recommendation

Reinstate the staleness protection code section in the oracle module of the Lending Market contract.

Status

This finding has been addressed by the client. Furthermore, the client has informed that the referred code section was commented out for testing purposes, and the contract was not designed to be deployed in Mainnet without this protection.

[A4] Silent Handling Of Failures In Mathematical Operations With unwrap_or(0) Can Corrupt The Protocol State

Severity: High | Difficulty: High | Recommended Action: Fix Code | Addressed by client

Description

Both the Lending Market and Collateral Vault contracts use `unwrap_or(0)` in critical calculations, which silently replaces overflow/underflow/division-by-zero with zero. For instance, in the Interest Rate Model's `get_r_comp`, many operations default to zero on overflow, including time delta, rate calculations, and compounding factors. For example, `delta_t = current_time.checked_sub(last_accrue_time).unwrap_or(0)` defaults to zero if the subtraction underflows, making the contract ignore time passage. Similarly, in another calculation within the IRM, `r_0 = r_i.checked_add(rp).unwrap_or(0)` and `r_1 = r_0.checked_add(slope.checked_mul(delta_t).unwrap_or(0)).unwrap_or(0)` can collapse to zero, causing the compounding factor to become 1.0 and stop interest accrual.

Fee calculations also use `unwrap_or(0)`. In `accrue_fee`, if fee calculations overflow, they default to zero, so fees are not collected. In the collateral vault, delegation and allocation math uses `unwrap_or(0)`, so overflows can zero out balances or totals. For instance, `new_receiver_total = current.checked_add(amount).unwrap_or(0)` can reset a borrower's delegated balance to zero on overflow, and `total.checked_add(amount).unwrap_or(0)` can reset global counters.

The IRM's `get_r_comp` chains many `unwrap_or(0)` calls, so a single overflow can cascade. For example, `slope_i = (params.k_i).checked_mul(u.checked_sub(params.u_opt).unwrap_or(0)).unwrap_or(0).checked_div(SCALE).unwrap_` can produce zero if any step overflows, breaking the rate calculation. When `delta_t` is zero due to underflow, subsequent multiplications by `delta_t` also become zero, making the IRM appear frozen. In some cases, catastrophic failures can occur, such as when calculating the new total amount of debt issued by the protocol within its compounding interest function (`compound_interest`, in `contracts/lending-market/src/irm.rs`), which can default to zero if an error is triggered during its calculation.

These silent failures can corrupt the protocol state. If interest stops accruing due to overflow defaults, borrowers benefit while lenders lose. If fee calculations default to zero, protocol revenue is lost. If delegation totals reset to zero, the vault's accounting breaks. If the amount of issued debt defaults to zero, the core logic of the lending market stops working (debt token price is zero, no liquidations, failure to repay debt, etc.). The contracts continue operating with wrong values instead of reverting, making these issues hard to detect and allowing incorrect states to persist.

Recommendation

Properly handle (either by panicking or executing correctional logic) the potential failures in mathematical operations throughout the protocol.

Status

This finding has been addressed in commit `a347810` by following the above recommendation. The client modified the instances of `unwrap_or` that pose a severe risk to their protocol, while accepting Runtime Verification's determination that the remaining ones are not a threat to the protocol's integrity.

[A5] Deallocation Incorrectly Increases `TotalDelegated`

Severity: High | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The collateral vault contract maintains a global counter `TotalDelegated` that should equal the total amount of funds delegated to borrowers (both allocated and unallocated). When a borrower calls `allocate`, the function moves funds from `Delegation` to `Allocation` without changing `TotalDelegated`, correctly preserving the total. However, when they call `deallocate`, the function erroneously increments `TotalDelegated` by the deallocation amount. This causes `TotalDelegated` to grow with each `deallocate` operation, even though the total delegated amount should remain unchanged.

Each call to `deallocate` irreversibly inflates `TotalDelegated`. Because no self-correcting mechanism exists, the deviation compounds over time with normal usage, and the contract has no path to recovering correct state without an upgrade or manual migration. The corrupted `TotalDelegated` state has two direct consequences on vault operations:

- **Artificial restriction of withdrawal capacity:** Since `max_withdraw` is derived from `TotalDelegated`, its value becomes progressively understated as the counter inflates. Over time, this can result in users being unable to withdraw funds they are legitimately entitled to, effectively locking tokens in the contract despite sufficient balance existing in the vault.
- **Denial of delegation capacity:** `remaining_capacity` is similarly affected, causing the vault to understate its actual available capacity. Consequently, the vault demands more collateral than is necessary to fulfill new delegations. To compensate for an inflation in `TotalDelegated`, the contract requires an equivalent amount of collateral to be deposited. As the inflation accumulates with each `deallocate` call, this excess collateral requirement grows over time.

Both consequences manifest over time through legitimate usage but can be accelerated by a malicious borrower deliberately cycling between `allocate` and `deallocate`, requiring no privileged access beyond having delegated any amount of collateral.

Scenario

The following steps demonstrate how `TotalDelegated` becomes corrupted:

```
1. The depositor deposits 500 tokens to the vault.
    * TotalDelegated = 0
    * remaining_capacity = 500
    * max_withdraw(depositor) = 500.

2. The delegator delegates 400 tokens to the borrower.
    * TotalDelegated = 400
    * remaining_capacity = 100
    * max_withdraw(depositor) = 100

3. The borrower allocates the 400 tokens to a market.
   * No change to TotalDelegated, capacity or max withdraw.

4. The borrower deallocates 400 tokens from the market.
   * TotalDelegated = 800
   * remaining_capacity = -300
   * max_withdraw(depositor) = -300
```

At this stage, the `TotalDelegated` value is corrupted, and withdrawal and delegation capacity is effectively broken.

```
5. The delegator undelegates 400 tokens.
   * TotalDelegated = 400
   * remaining_capacity = 100
   * max_withdraw(depositor) = 100
```

After step 5, all active delegations have been fully undelegated. Therefore:

- `remaining_capacity` should equal the initial capacity of 500, since no funds are actively delegated.
- The depositor should be able to withdraw their full 500 tokens.

However, due to the previously inflated `TotalDelegated` value, only 100 tokens are withdrawable, while the remaining 400 tokens remain locked in the contract despite no outstanding delegation existing. Moreover, repeated cycling of steps 3 and 4 continues to inflate `TotalDelegated`, progressively degrading vault behavior and increasing the gap between real economic state and recorded accounting state.

Recommendation

The fix is to remove the statement that increases `TotalDelegated` in the `deallocate` function. Since `deallocate` only moves amounts from `Allocation` to `Delegation` without changing the total delegated funds, `TotalDelegated` should remain unchanged, matching the pattern used in `allocate`.

Status

This issue has been resolved in commit `fce3f77`, with the fix implemented according to the recommendation.

[A6] Lending Market Allows Withdrawing More Collateral Than Deposited

Severity: High | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The `withdraw_collateral` function does not validate that the requested withdrawal amount does not exceed the borrower's current collateral balance. The function calculates the final collateral balance using `checked_sub`. However, since the values are `i128`, `checked_sub` only returns `None` when the result overflows the bounds of `i128`, which is never the case in practice. This means a negative result is returned without any error, and there is no explicit validation that `amount <= current_collateral` to enforce the business logic constraint that a borrower cannot withdraw more than their deposited collateral.

When delegation is enabled, the `min_collateral` check prevents this by ensuring the collateral does not fall below a valid threshold, which is a non-negative number. However, when delegation is disabled, no such check exists, allowing the collateral balance of the account to go negative. A borrower can exploit this to withdraw far beyond their deposited collateral, potentially draining all liquidity available in the lending market.

Scenario

The following is the list of theoretical steps exemplifying this exploit:

```
0 - Initially, the lending market holds 100 tokens of collateral and delegation is disabled.
1 - A malicious user with 0 collateral calls `withdraw_collateral` with amount X.
2 - The function computes `new_amount = 0 - 100`, returning a negative value with no error.
3 - The contract transfers 100 tokens to the malicious user.
4 - The negative `new_amount` is stored as the borrower's collateral balance.
```

Recommendation

Add an explicit check that the requested withdrawal amount does not exceed the borrower's current collateral balance before performing the subtraction.

Status

This issue has been resolved in commit `81915a2`, with the fix implemented according to the recommendation.

[A7] Incorrect Operation Order in Liquidation Threshold Check Causes Precision Loss

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The `liquidation_threshold_bps` check in the `withdraw_collateral` function is intended to compare the ratio of `borrowed` to `new_principle_value` against the configured liquidation threshold. Since all values are integers, borrowed is multiplied by `10000` to scale the ratio to basis points, preserving precision for a meaningful comparison against `liquidation_threshold_bps`. However, the implementation performs the division first and then multiplies by `10000`.

Integer division truncates any fractional result, so the precision is lost before the scaling is applied. Any ratio below `1` is truncated to `0`, and multiplying `0` by `10000` still yields `0`, rendering the check ineffective. As a consequence, undercollateralized positions cannot be liquidated, and the protocol is left exposed to bad debt.

Recommendation

Multiply borrowed by `10000` first to preserve precision before dividing by `new_principle_value`.

Status

This issue has been resolved in commit `81915a2`, with the fix implemented according to the recommendation.

[A8] Lending Market `withdraw_collateral` Calculates Inverted Allocation Ratio

Severity: High | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Description

The `max_allocation_rate` check in the `withdraw_collateral` function is supposed to implement the same allocation rate limit that is enforced in the collateral vault's `allocate` and `deallocate` functions. The check is intended to compare the ratio of collateral allocated to the lending market against the direct collateral held in the lending market. However, the numerator and denominator are swapped — the check computes direct collateral divided by allocated collateral instead of the inverse. As a consequence, borrowers are blocked from withdrawing their excess collateral even when they legitimately should be able to, since the inverted ratio causes valid withdrawal attempts to fail the check.

Recommendation

Swap the numerator and denominator in the allocation ratio calculation to match the correct ratio used in the collateral vault's `allocate` and `deallocate` functions.

Status

This issue has been resolved in commit `81915a2`, with the fix implemented according to the recommendation.

[A9] Precision Loss in Integer Division Leads To Misconfiguration And Collateral Drainage

Severity: High | Difficulty: High | Recommended Action: Fix Code | Addressed by client

Description

Regarding the mathematical operations of the Lending Market, several of them are prone to precision loss (detrimental to the protocol). For a start, many of the operations related to manipulating the core protocol state rely on `get_borrowed_value` for its calculations, which uses a floor division `floor(borrowed_in_debt_token * debt_token_price / 10^7)`. This function is the single source of truth for a borrower's debt across every solvency-critical path in the protocol, and it reduces the debt amount by flooring this calculation, ever so slightly, but always. Besides compounding the precision loss over time, it directly affects the calculations in `is_healthy()`, `available_liquidation()`, `get_borrow_capacity()`, `withdraw_collateral()`, and `validate_deallocate()`, all of which are fundamental operations for managing borrowed amounts.

This behavior is repeated across other operations of the Lending Market, like the `borrow()`, which calculates `debt_token_amount`. Every time you floor this division and the precision loss is manifested, you're computing less debt for the amount that has actually been deposited as collateral.

The rounding on `borrow()` has the highest potential for exploits. As unlikely as it is, if `amount x 10^7 < debt_token_price` and the borrow token value is high enough for the exploit to be feasible (profit higher than cost), the borrower gets to borrow the provided amount while issuing 0 debt shares. The exploit can be repeated, and liquidity would eventually be drained from the Lending Market.

Investigations suggest that the protocol rounds calculations in its detriment in `liquidate()` (first when calculating the collateral taken from borrower, and second when calculating the debt cleared by the liquidator), `convert_to_assets()` when called through `mint`, `convert_to_shares` when called through `withdraw()`, `calc_fee()` and `calc_fee_assets`, though none of these should have an impact as meaningful as the issues in `borrow` and `get_borrowed_value`.

Nuances Of The Precision Loss In The IRM

The Interest Rate Model (IRM) compounds interest by computing new debt as the old debt plus an interest increment, applying an integer floor division by a fixed scaling constant (`new_total_debt_assets = floor(total_debt_assets * (r_comp + SCALE) / SCALE)`, where `SCALE = 10^18`). Because the protocol operates in fixed-point arithmetic, any fractional interest below 1 native token unit is truncated to zero. A 5% annual rate over 5 seconds (one Stellar ledger time) covers a tiny fraction of the year (5 out of 31,536,000 seconds), producing a growth factor on the order of `10^9`, which is roughly a billion times smaller than SCALE. To offset this disparity, the debt must be on the order of a billion native units for interest to survive the division. This evaluates to approximately 126 million native units, which, for a 7-decimal token, is equivalent to 12.62 tokens. For standard Stellar tokens like USDC, this $12.62 threshold is negligible, and any realistic pool exceeds it. However, because the threshold is constant in native units, its dollar value scales inversely with token decimals: approximately $1,261 for 5-decimal tokens and $1,261,440 for 2-decimal tokens. A lending pool for a 2-decimal token with over one million dollars in outstanding debt would still accrue zero interest under per-ledger compounding. High-value tokens may suffer from the same issue, despite having a higher number of decimals as well.

For pools above the threshold, the cumulative truncation loss is bounded by the number of compounding cycles per year multiplied by a maximum per-cycle loss of 1 native unit, which is a fixed cost independent of pool size. At worst-case per-ledger compounding, this amounts to $0.63/year for 7-decimal tokens, but over $63,000 for 2-decimal tokens. In practice, compounding only occurs when a user interacts with the protocol (deposit, withdraw, borrow, or repay), so the actual interval between accruals is typically larger and the loss proportionally smaller.

For pools below the threshold, interest is completely suppressed as borrowers pay nothing and lenders earn nothing. The current implementation unconditionally resets the last accrual timestamp every time compounding is triggered, regardless of whether interest

was actually produced. This prevents elapsed time from accumulating across cycles, making the suppression permanent.

Recommendation

To address these issues in the scenarios and functions described above, we recommend using ceiling division rather than floor division and rounding the calculated values in accordance with the protocol health, except in the IRM.

Considering the interest calculations, ceiling divisions would ensure that at least 1 native unit of interest is always accrued, but it would introduce the opposite problem: the protocol would consistently overcharge interest, minting debt out of thin air. Therefore, for the IRM, we do not recommend modifying the rounding operation directions. A targeted fix is to make the state updates in the compounding function conditional on interest actually being accrued. Instead of unconditionally updating the rate model state and the last accrual timestamp, which is done within the `compound_interest()` function, these writes should only occur when the new debt exceeds the old debt. This allows elapsed time to accumulate naturally across zero-interest cycles until the compounding factor yields a nonzero result, at which point all accumulated interest is applied at once and the timestamp resets.

Additionally, the team should document that, if the above fix is implemented, users of Lending Markets handling tokens with low decimal counts or low total debt will likely experience delays in the visualization of interest accrual, which are unavoidable due to the limitations of handling precision loss from division in small numbers.

Status

The finding has been addressed in commits `a347810`, `5743057`, and `6b6d473`, which appropriately modify the division rounding and follow the recommendations above.

[A10] Possible Denial-of-Service Caused By Storage Limitations

Severity: Medium | Difficulty: Medium | Recommended Action: Fix Code | Addressed by client

Description

The Lending Market and Collateral Vault contracts store all per-user and per-event data (borrower debt balances, collateral amounts, pending liquidation claims, delegation entries, and liquidation receipts) in Soroban instance storage. Instance storage packs every key-value pair into a single ledger entry, which is subject to a protocol-enforced size limit (currently 128 KiB). Each new user or liquidation event permanently adds entries to this shared space, and certain keys (such as `PendingSeize` and `IsLiquidated`) are never deleted after fulfillment.

This means the contracts have a hard ceiling on how many participants and events they can accommodate. Under normal organic growth, a lending market reaching a few hundred to a thousand active users would approach the storage limit, depending on the number of liquidations that happen in the meantime. Once the limit is reached, any operation that attempts to write a new key (a new borrower supplying collateral, a liquidation recording a `PendingSeize` entry) will fail. The contract does not degrade gracefully, as it simply stops accepting new participants and may be unable to execute liquidations at this point.

Recommendation

The standard Soroban practice for unbounded per-user data is to use persistent storage (`e.storage().persistent()`), where each key occupies its own independent ledger entry with no shared size constraint. Migrating user-scoped and event-scoped keys to persistent storage would remove the scalability ceiling and allow the protocol to grow without an upper bound on participants.

Status

This finding has been addressed in commit `ed01bfa`. The client remediated the finding by converting all unbounded data to persistent storage.

[A11] Incorrect `Delegation` Update in `liquidate_allocation`

Recommended Action: Fix Code | Addressed by client

Description

This is a follow-up issue introduced while fixing the *[A2] Liquidations in the Collateral Vault Don't Update* `TotalAllocated` issue.

The original fix correctly added logic to decrement `TotalAllocated` during `liquidate_allocation`. It also added updates to `TotalDelegated`, which is appropriate because `TotalDelegated` includes both delegated and allocated amounts.

However, the fix additionally decrements the borrower's **individual** `Delegation` amount. This update is incorrect.

During the normal flow, when a borrower calls `allocate`, the vault decreases the borrower's `Delegation` and increases their `Allocation`. Therefore, by the time liquidation occurs, the delegated collateral corresponding to the allocation has already been removed from the borrower's delegation.

When `liquidate_allocation` reduces the borrower's `Allocation`, it should **not** also reduce their `Delegation`, as this would double-count the reduction.

The correct accounting during liquidation should be:

- **Individual allocation**: decrease by `seize_amount`
- **TotalAllocated**: decrease by `seize_amount`
- **Individual delegation**: no change
- **TotalDelegated**: decrease by `seize_amount` (because it includes allocated amounts)

Updating the borrower's `Delegation` during liquidation breaks the intended accounting model and may lead to incorrect delegation balances.

Recommendation

Remove the update to the borrower's `Delegation` in `liquidate_allocation`.

Status

This issue has been resolved in commit `6b6d473`, with the fix implemented according to the recommendation.

Informative Findings

This section includes observations that are not directly exploitable, but highlight areas for improvement in code clarity, maintainability, or best practices. While not critical, addressing these can strengthen the system's overall robustness.

[B1] Best Practices Recommendations

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

Here are some notes on the protocol's particularities, comments, and suggestions to improve the code or the business logic of the protocol in a best-practice sense. They do not present issues with the audited protocol themselves. Still, they are advised to either be aware of or to follow when possible, and they may explain minor unexpected behaviors in the deployed project.

1. Operations performing important state modifications may not emit events in the Lending Markets. For example, `set_delegate_enabled` in `market.rs` toggles if delegated collateral is considered in health calculations, yet no event is emitted.
2. Throughout the codebase, decimal scaling factors such as `10i128.pow(7)` and `10_000` (for basis points) are hardcoded inline in arithmetic expressions across multiple files. For instance, `10i128.pow(7)` appears in `borrow.rs`, `liquidation.rs`, and `irm.rs` for debt token price conversions.
3. In the Lending Market, core calculations such as price conversion using oracle data (with decimal normalization) and loan-to-value (LTV) computation are implemented repeatedly using inline arithmetic rather than shared utilities. This spreads critical economic logic across the codebase, increasing the risk of inconsistencies and making the system harder to maintain or audit.
4. In multiple places in the codebase, `checked_div` is used when dividing by compile-time non-zero constants or values otherwise guaranteed to be non-zero (e.g. powers of 10). Since `checked_div` only guards against division by zero, these usages add unnecessary complexity without improving safety.
5. Ceiling division is implemented in multiple places using the pattern `(a + b - 1) / b` through chains of checked arithmetic operations (`checked_add`, `checked_sub`, `checked_div`). This results in error-prone and difficult-to-read expressions and duplicates the same arithmetic logic across the codebase.

Recommendation

For each of the topics elaborated above, we recommend implementing the following approaches into the protocol's contracts:

1. Emit events in all administrative functions that modify protocol state, including `set_delegate_enabled`, to maintain observability and allow off-chain systems to react to configuration changes.
2. Define named constants for all recurring decimal scaling factors and basis point denominators. Replace all inline occurrences of `10i128.pow(7)`, `10_000`, and similar values with their corresponding constants. This reduces the probability of mistakenly using different decimal values at different points of the code and improves readability.
3. Centralize core financial computations, such as oracle price conversion and LTV calculation, into dedicated internal helper functions and reuse them throughout the contract.
4. Replace `checked_div` with regular division (`/`) when the divisor is guaranteed to be non-zero.
5. Encapsulate the ceiling division logic into a dedicated helper function (e.g., `ceil_div`) to avoid repeating the `(a + b - 1) / b` pattern across the codebase and significantly improve readability.

Status

Topics 2 and 5 have been addressed in commit `6b6d473`, while the remaining points not explicitly integrated into the code are simply stylistic choice suggestions.

[B2] Mismatching Error Messages For Specific Behaviors

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

Error variants across the protocol's contracts are reused or misnamed, causing misleading error messages that do not accurately reflect the condition that triggered them. This can hinder debugging, confuse integrators, and make it harder for users and off-chain systems to programmatically identify and handle specific failure cases.

1. In the `flash_loan` function of the Lending Market (`contracts/lending-market/src/market.rs`), when the provided `token` address does not match the market's underlying asset, the function panics with `LendingMarketError::ZeroFlashLoanAmount` . This error is misleading, as the issue is not a zero amount but an invalid token.
2. In the Collateral Vault, the `undelegate` function in `contracts/collateral-vault/src/delegation.rs` triggers `VaultError::AllocationExceedsDelegation` when the caller attempts to undelegate more than the current delegation balance. The error name suggests an allocation operation is being performed, when in fact it is an undelegation.
3. In the Collateral Vault, the `deallocate` function in `contracts/collateral-vault/src/delegation.rs` triggers `VaultError::MaxAllocationRatioExceeded` when a partial deallocation would leave a ratio-violating remainder. The error name suggests an allocation is being increased past a cap, which is counterintuitive during a deallocation operation.
4. In the Lending Market, the `withdraw_collateral` function in `contracts/lending-market/src/borrow.rs` uses a single `LendingMarketError::CollateralTooLow` error for two distinct failure conditions: the borrower's collateral dropping below `min_collateral` , and the allocation-to-collateral ratio exceeding `max_allocation_rate` . The second condition is a ratio constraint violation, not a "collateral too low" problem.

Recommendation

For each of the topics elaborated above, we recommend implementing the following approaches into the protocol's contracts:

1. Introduce a dedicated error variant (e.g., `InvalidFlashLoanToken`) for the token mismatch check in the `flash_loan` function and replace the reuse of `ZeroFlashLoanAmount` .
2. Rename the `AllocationExceedsDelegation` error variant to a name that accurately describes the undelegation context, such as `UndelegationExceedsDelegation` , and use it in the `undelegate` function.
3. Introduce a dedicated error variant for ratio constraint violations during deallocation (e.g., `DeallocationRatioViolation`) to distinguish it from the allocation-side `MaxAllocationRatioExceeded` .
4. Split the `CollateralTooLow` error in `withdraw_collateral` into two distinct variants: one for the minimum collateral threshold (e.g., `CollateralBelowMinimum`) and one for the allocation ratio constraint (e.g., `AllocationRatioExceeded`).

Status

This finding has been addressed in commit `81915a2` by following the recommendations above.

[B3] Implicit Reliance on Arithmetic Fallback Values for Control Flow Correctness

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

This finding is an addendum to A4, which addresses the use of `unwrap_or` operations throughout the protocol.

Critical validation paths in the protocol reach the correct outcome not through explicit, intentional logic, but through the side effects of arithmetic error handling, specifically the `unwrap_or(0)` pattern in `checked_div` operations. The most notable instance is the delegation constraint check in `withdraw_collateral` (`contracts/lending-market/src/borrow.rs`). When a borrower has no delegated collateral (`allocated == 0`), the division `new_amount × 10_000 / allocated` triggers a division by zero. Rather than being caught by an explicit guard (e.g., `if allocated == 0 { skip check }`), the `unwrap_or(0)` silently converts the error into `0` , which then fails the `> max_allocation_rate` comparison, allowing the withdrawal to proceed. The check produces the correct result: a borrower with no delegation should not be constrained by delegation rules, but it does so accidentally, relying on the seeming coincidence that `0` happens to be the permissive value in this comparison.

This pattern is fragile because it couples correctness to an implicit assumption about what `0` means downstream. If the comparison operator or the semantics of `max_allocation_rate` were ever changed (for instance, if `0` were treated as "no allocation allowed" rather than a permissive default), the behavior would silently invert without any explicit logic having been modified. The same `unwrap_or(0)` pattern appears broadly across the codebase's arithmetic: in `get_borrowed_value()` , `get_principle_value()` , `get_borrow_capacity()` , and throughout the IRM module. In most of these cases, a `0` result on overflow or division error happens to be safe (e.g., zero debt means the position appears healthy, zero capacity means no borrowing is allowed). But the safety is coincidental rather than assured; each instance requires the reviewer to independently verify that `0` is the correct fallback in that specific context, and any future refactoring could break these implicit assumptions without triggering any compile-time or runtime warning.

Recommendation

A more robust approach would be to handle edge cases explicitly before performing the arithmetic. For the delegation check, an `if allocated == 0 { return }` guard would make the intent clear and remove the dependency on `unwrap_or(0)` , producing a safe value. More broadly, the codebase would benefit from distinguishing between arithmetic operations where failure is a genuine protocol error (which should panic with a typed error) and those where a default value is intentionally appropriate (which should be documented as such). The current uniform use of `unwrap_or(0)` obscures this distinction, making it difficult to determine whether a zero fallback at any given point is a deliberate design choice or an overlooked edge case.

Status

This issue has been resolved in commit `ac7cca9` , with the fix implemented according to the recommendation.

[B4] Finalizing the Oracle Address Update Requires Admin Intervention, Allowing Stale Oracle Usage

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

The Lending Market uses a timelock for oracle updates. When an admin calls `set_oracle` to propose a new oracle, the contract stores the new address as pending and sets an execution timestamp 48 hours in the future. The update does not happen automatically. The admin must call `execute_oracle_update` after the timelock expires to switch from the old oracle to the new one. This requires the admin to monitor the timelock and call `execute_oracle_update` within a reasonable window after it expires.

The issue is that price fetching does not check for pending oracle updates. When any function calls `get_price` (e.g., during borrows, liquidations, or collateral value calculations), it reads from the current oracle address stored in the contract state. It does not check whether a pending oracle update is ready to execute. As a result, the market can continue using an outdated oracle even after the timelock expires, until the admin manually calls `execute_oracle_update`. This creates a window where stale or incorrect prices may be used for critical operations.

Recommendation

A better design would embed the oracle update check inside the price-fetching module. Whenever `get_price` is called, it would first check if there is a pending oracle update and whether the timelock has expired. If so, it would automatically execute the update before fetching the price. This ensures the market always uses the most recent approved oracle without relying on manual admin intervention. It also removes the risk of accidentally querying an outdated oracle after the timelock period, since any price fetch operation would trigger the update check automatically.

Status

This issue has been resolved in commit `6b6d473`, with the fix implemented according to the recommendation.

[B5] Single Parameter Check Blocks All Market Config Setters, Possibly Causing Partial Configuration Of Lending Market

Severity: Informative | Recommended Action: Fix Code | Addressed by client

Description

The lending market uses `is_market_config_set` to gate three admin setters: `set_liquidation_threshold`, `set_liquidation_bonus_bps`, and `set_market_config`. However, `is_market_config_set` only checks whether the `LiquidationThreshold` storage key exists. Once `set_liquidation_threshold` is called, `is_market_config_set` returns true, blocking all three functions. This makes `liquidation_threshold` an implicit initialization flag: once set, the other parameters cannot be set afterward, even if they were never initialized.

This creates a partial configuration problem. If an admin calls `set_liquidation_threshold` first, the threshold is set, but `liquidation_bonus_bps` remains at its default and cannot be set afterward. The market then operates with a liquidation threshold but no liquidation bonus, which breaks liquidation incentives. Liquidators expect a bonus to compensate for risk and gas costs; without it, liquidations may not occur, leaving unhealthy positions in the system. The combined `set_market_config` function also becomes unusable, forcing the admin to set parameters individually, but only the first one can be set.

Recommendation

Change `is_market_config_set` to check that all three parameters (liquidation threshold, liquidation bonus, and fee) are set before blocking further changes. Alternatively, allow updates after initial configuration by removing the gate or adding an update path.

Status

This issue has been resolved in commit `fce3f77`. The separate admin setters `set_liquidation_threshold`, `set_liquidation_bonus_bps`, and `set_fee` were removed, leaving `set_market_config` as the single configuration entry point for these parameters. By consolidating initialization into one function, the configuration can no longer enter a partially initialized state.

[B6] Lending Markets Allow Collateral And Borrowing Assets To Be The Same

Severity: Informative | Recommended Action: Fix Design | Addressed by client

Description

The Lending Market constructor does not validate that the borrowed asset and collateral asset are different.

If a market is deployed with the same token address for both, the protocol does not revert or prevent the deployment, proceeding normally instead. However, the health check logic was not designed for this scenario: the collateral side is valued through an external oracle while the debt side is valued through an internal price mechanism, meaning both sides of the same-asset position respond differently to the same market conditions. This mismatch can produce unpredictable behavior in the health check, liquidation triggers, and borrowing capacity calculations, none of which were designed or tested under the assumption that both assets are identical. Adding a guard in the constructor that requires the lending asset and collateral asset to have different addresses would eliminate this class of misconfiguration at deployment time.

Recommendation

To prevent possible loss of assets by unpredictable behavior caused by having the same asset as collateral and borrowing asset, in the constructor of the Lending Market, enforce that `asset` is different from `collateral`.

Status

This finding has been addressed in commit `6ef5004` by following the above recommendation.