

Security Assessment Report



Fluid Vaults

October 2026

Prepared for Fluid / Mysten Labs

certora

Table of Contents

Project Summary **4**
- Project Scope 4
- Project Overview 4
- Protocol Overview 4

Assessment Methodology **6**

Threat and Security Overview **7**
- Findings Summary 10
- Severity Matrix 10

Detailed Findings **11**
- Medium Severity Issues 14
  - M-01: Premature debt rounding can block liquidation of unsafe positions 14
  - M-02: Partial branch ratio falls below the next perfect tick and blocks liquidation 17
- Low Severity Issues 19
  - L-01: Fully liquidated positions cannot be closed without new funded activity 19
  - L-02: Withdrawal-gap check floors a share conversion that Liquidity ceils, so one raw unit of the buffer can be withdrawn 21
  - L-03: Zero-collateral liquidation reconstruction blocks collateral-only recovery 23
  - L-04: Combined repayment and withdrawal reverts above CF despite reducing risk 26
  - L-05: Unfunded borrow fee reduces depositor yield 28
  - L-06: Missing dust-debt reconciliation after full liquidation understates total_borrow 30
  - L-07: A position accepted at the LT limit tick can be liquidated at the same price 32
  - L-08: Over-debiting one position can clear a shared tick and block another borrower's exit 35
  - L-09: Deposit and partial repay require an operate-grade oracle quote even though the health check only uses the liquidate rate 37
- Informational Severity Issues 39
  - I-01: Cap-free deposit re-fills supply_amount, griefing the empty-position gate in position::delete 39
  - I-02: LML validation rejects inputs that become valid after truncation 40
  - I-03: Vault configuration events omit the affected vault and use inconsistent risk-parameter scales 41
  - I-04: Incomplete initialization validation permits unusable vault markets 43
  - I-05: Withdraw gap skipped when liquidity limit is 0 in check_if_withdrawal_safe_for_withdrawal_gap 45
  - I-06: Magnifier rate_change floors on negative supply and positive borrow, underpaying the rebalancer by 1 wei 46
  - I-07: LogUserPosition reports gross tick debt instead of the user's net borrow 48
  - I-08: Narrow LT–LML settings can eliminate ordinary liquidation 49
  - I-09: emit_log_liquidation_rounding_diff event is emitted on a path that always aborts 50
  - I-10: Unfunded borrow fee prevents full repayment 51
  - I-11: Redundant bitmap clear when the settled tick is the top tick 53
  - I-12: Relative liquidation minimum rounds down by one debt share 54

I-13: LogInitVaultState does not include the initial pool exchange prices....55
Disclaimer....56
About Certora....56

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Fluid Vaults	https://github.com/Instadapp/fluid-packages-sui	Initial: fc6fad6 Final: 9a1bb2	Sui / Move

Project Overview

This document describes the security review of **Fluid Vaults**. The work was undertaken from **September 10th, 2026** to **October 9th, 2026**.

The following contract list is included in our scope:

packages/vaults/sources/**/*.move
packages/library/sources/**/*.move

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

Vaults is Fluid's collateralized borrowing layer on the Liquidity Layer. Each vault pairs one collateral asset with one debt asset. Users open a position, deposit collateral, and borrow against it; the vault supplies collateral into the shared pool and draws debt from it. A transferable position owner cap authorizes borrowing and withdrawals, while anyone can add collateral or repay debt. Supply and borrow exchange prices track accrued interest, with configurable vault rate adjustments. Oracle prices and risk limits determine how much users can borrow or withdraw.

Positions are grouped into ticks by debt-to-collateral ratio. When positions cross the liquidation threshold, anyone can repay debt in exchange for collateral plus a liquidation incentive.

Liquidation works through the riskiest ticks first, with branches tracking shared liquidation progress across affected positions. Positions beyond the maximum liquidation limit are absorbed into separate vault accounting for later settlement. Admins create markets, connect them to liquidity, and configure oracles, rates, fees, and risk limits. Authorized rebalancers reconcile differences between the vault's recorded collateral and debt and its actual Liquidity Layer positions.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

certora

Threat and Security Overview

The Vaults Layer provides collateralized borrowing through the shared Liquidity Layer. Each `VaultMarket<Col, Debt>` manages collateral and debt positions, derives exchange prices from underlying reserves, and organises liquidation accounting through ticks and branches. Users control withdrawals and borrowing through `PositionOwnerCap` objects, while permissionless participants can deposit, repay, update exchange prices, and liquidate eligible positions. Privileged roles configure markets, oracle selection, risk parameters, and rebalancing. Its main security boundaries are collateral sufficiency, position authority, oracle integrity, liquidation correctness, and consistency with underlying liquidity accounting. The following threat model summarises the assets, actors, assumptions, and non-exhaustive attack vectors list considered during review.

Assets

- **User collateral** - supplied assets held through the Liquidity Layer.
- **Borrower positions** - principal, accrued debt, and borrowing fees.
- **Position ownership** - authority to withdraw collateral, borrow, and close positions.
- **Vault liquidity capabilities** - embedded `UserCap` objects controlling reserve positions.
- **Exchange-price state** - supply and debt indices determining position balances.
- **Liquidation state** - ticks, bitmaps, branches, and historical liquidation snapshots.
- **Absorbed balances** - collateral, debt, and dust tracked after absorption.
- **Rebalancing balances** - surpluses and funding requirements between vault and reserve accounting.
- **Risk configuration** - collateral factors, liquidation limits, penalties, and withdrawal gaps.

Actors

- **Protocol governance** - manages administrative authority, upgrades, and registry migration.
- **Auth operators** - configure markets, oracles, rates, fees, and risk limits.
- **Position owners** - hold capabilities authorizing withdrawals and borrowing.

- **Third-party depositors and repayers** - add collateral or repay debt without ownership authority.
- **Permissionless liquidators** - repay eligible debt in exchange for collateral.
- **Permissionless keepers** - refresh vault exchange prices.
- **Authorized rebalancers** - reconcile vault accounting with underlying reserve positions.
- **Oracle administrators and providers** - configure and supply valuation data.
- **Liquidity Layer operators** - manage underlying reserve permissions, limits, and pauses.
- **Coin issuers** - control token issuance and applicable transfer restrictions.

Trust Assumptions

- **Capability custody** - owners and integrations protect position capabilities.
- **Oracle correctness** - quotes reflect timely, correctly scaled collateral-to-debt valuations.
- **Liquidity Layer integrity** - reserves preserve custody, accounting, and capability enforcement.
- **Supported reserve modes** - underlying positions use interest modes compatible with vault accounting (only WI mode is supported).
- **Safe configuration** - token decimals, risk limits, fees, and rate magnifiers remain appropriate.
- **Liquidation liveness** - liquidators have sufficient incentives, funding, and transaction access.
- **Rebalancing capacity** - authorized operators can fund accounting shortfalls when required.
- **Token availability** - collateral and debt coins remain transferable.
- **Sui correctness** - object ownership, transaction atomicity, clocks, and arithmetic semantics hold.
- **Upgrade integrity** - upgrades preserve position rights and accounting invariants.

Attack Vectors

- **Position-capability abuse** - theft, sharing, freezing, or loss compromises position control.
- **Cross-market substitution** - mismatched positions, capabilities, registries, or quotes bypass intended bindings.
- **Multi-leg authorization bypass** - combined operations obscure withdrawals or borrowing requiring ownership.
- **Oracle manipulation or staleness** - incorrect valuations enable excessive borrowing or unfair liquidation.

- **Decimal and unit mismatches** - inconsistent scaling distorts collateral checks, settlement, or slippage protection.
- **Tick-boundary manipulation** - rounding and tick placement misclassify debt or liquidation eligibility.
- **Branch-accounting divergence** - merges, debt factors, or historical snapshots misallocate liquidation losses.
- **Lazy position reconciliation** - stale position state restores collateral or debt already liquidated.
- **Absorption-accounting errors** - deferred settlement omits, duplicates, or strands absorbed balances.
- **Dust and minimum-amount constraints** - residual balances obstruct repayment, withdrawal, or liquidation.
- **Precision-loss cycling** - repeated operations accumulate favorable rounding discrepancies.
- **Exchange-price divergence** - rate magnifiers create obligations unsupported by underlying reserve positions.
- **Rebalancing abuse** - incorrect settlement direction or compromised authority extracts excess assets.
- **Liquidity exhaustion or pauses** - reserve restrictions prevent exits or timely liquidation.
- **Liquidation execution exhaustion** - extensive tick scans or branch traversal exceed transaction limits.
- **Privileged configuration abuse** - unsafe oracle or risk changes enable extraction or abrupt liquidations.
- **Market gaps and bad debt** - rapid price movements outpace liquidation and leave uncovered liabilities.
- **Shared-liquidity contagion** - losses or shortages elsewhere in the Liquidity Layer impair vault operations.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	2	2	2
Low	9	9	2
Informational	13	13	4
Total	24	24	8

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

certora

Detailed Findings

ID	Title	Severity	Status
M-01	Premature debt rounding can block liquidation of unsafe positions	Medium	Fixed
M-02	Partial branch ratio falls below the next perfect tick and blocks liquidation	Medium	Fixed
L-01	Fully liquidated positions cannot be closed without new funded activity	Low	Acknowledged
L-02	Withdrawal-gap check floors a share conversion that Liquidity ceils, so one raw unit of the buffer can be withdrawn	Low	Acknowledged
L-03	Zero-collateral liquidation reconstruction blocks collateral-only recovery	Low	Acknowledged
L-04	Combined repayment and withdrawal reverts above CF despite reducing risk	Low	Fixed
L-05	Unfunded borrow fee reduces depositor yield	Low	Acknowledged
L-06	Missing dust-debt reconciliation after full liquidation understates total_borrow	Low	Acknowledged
L-07	A position accepted at the LT limit tick can be liquidated at the same price	Low	Acknowledged

L-08	Over-debiting one position can clear a shared tick and block another borrower's exit	Low	Acknowledged
L-09	Deposit and partial repay require an operate-grade oracle quote even though the health check only uses the liquidate rate	Low	Fixed
I-01	Cap-free deposit re-fills supply_amount, griefing the empty-position gate in position::delete	Informational	Acknowledged
I-02	LML validation rejects inputs that become valid after truncation	Informational	Acknowledged
I-03	Vault configuration events omit the affected vault and use inconsistent risk-parameter scales	Informational	Partially Fixed
I-04	Incomplete initialization validation permits unusable vault markets	Informational	Fixed
I-05	Withdraw gap skipped when liquidity limit is 0 in check_if_withdrawal_safe_for_withdrawal_gap	Informational	Acknowledged
I-06	Magnifier rate_change floors on negative supply and positive borrow, underpaying the rebalancer by 1 wei	Informational	Acknowledged
I-07	LogUserPosition reports gross tick debt instead of the user's net borrow	Informational	Acknowledged
I-08	Narrow LT–LML settings can eliminate ordinary liquidation	Informational	Fixed

I-09	emit_log_liquidation_rounding_diff event is emitted on a path that always aborts	Informational	Fixed
I-10	Unfunded borrow fee prevents full repayment	Informational	Acknowledged
I-11	Redundant bitmap clear when the settled tick is the top tick	Informational	Acknowledged
I-12	Relative liquidation minimum rounds down by one debt share	Informational	Acknowledged
I-13	LogInitVaultState does not include the initial pool exchange prices	Informational	Acknowledged

Medium Severity Issues

M-01: Premature debt rounding can block liquidation of unsafe positions

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: packages/vaults/sources/utils/liquidate.move	Status: Fixed	

Description:

The liquidation loop processes the highest branch or tick before moving to the next lower reference tick. For each interval, `get_debt_liquidated()` calculates the repayment needed to reach that reference. When the current item is a branch, the lower tick's debt has not been added to the calculation yet.

The helper floors the projected debt before subtracting it from the current debt:

```
let numerator =
    (self.debt - self.get_debt_from_ratios())
        * constants::rate_output_precision();
let denominator =
    constants::rate_output_precision()
        - self.get_col_from_ratios(col_per_debt);
numerator / denominator
```

`get_debt_from_ratios()` returns `floor(debt × reference_ratio / current_ratio)`. If the projected debt is fractionally below the current debt, flooring it first can turn a fractional difference into one full raw debt share. Division by the liquidation denominator can then produce a repayment greater than the branch's entire debt. This is reachable after a small position is partially liquidated into a branch and a further price drop makes that branch and a lower, untouched tick liquidatable.

Consider the scenario in the PoC, which uses nine-decimal debt. A large loan sits at a lower tick, while an attacker controls a much smaller position at a higher tick. The first price drop makes only the small position liquidatable; partially liquidating it leaves a branch. The second drop

makes both the branch and the untouched large position eligible for ordinary liquidation, without triggering absorption. The loop processes the branch first and uses the large position's tick as its reference.

```
Large loan                              71,300 debt tokens, tick −225
Small loan                              1,000 native debt atoms
After first drop to 0.841               Branch at tick −223, debt 102 raw shares
After second drop to 0.754              Branch and large tick both liquidatable
```

For the branch's interval, the calculation is:

```
Branch debt (D)                         102
Branch ratio (R)                        201199167236491
Lower tick's reference ratio (r)        200897820505453
Liquidation denominator (B)             8915135004786

Projected debt at the reference         D × r / R ≈ 101.847229
Projected debt after early flooring     101
Current repayment calculation           floor((102 − 101) × 10¹⁵ / B) = 112
Without the early floor                 floor(D × (R − r) × 10¹⁵ / (R × B)) = 17
```

The caller adjusts the result only when it equals the current debt:

```
if (current_data.debt() == debt_liquidated) {
    debt_liquidated = debt_liquidated - 1;
};
```

Here, the calculated repayment exceeds the branch's debt, so the check does nothing. `get_debt_factor()` attempts `102 − 112` and aborts the transaction before the large position can be liquidated. An attacker who also controls that position can use the small one to delay liquidation of their own unsafe loan at the affected prices. A later price change may end the blockage or make the branch absorbable.

The PoC expects this arithmetic abort. Save the linked file as `packages/vaults/tests/rounding_dos_tests.move` in a `feat/vault-protocol` checkout at `fc6fad65`, then run from the repository root:

```
sui move test --path packages/vaults --build-env mainnet
test_partial_branch_blocks_large_position_liquidation
```

Recommendations:

Keep the same underlying liquidation formula, but preserve precision until the final division. Update `get_debt_liquidated` as follows:

```
public fun get_debt_liquidated(
    self: &CurrentLiquidity,
    col_per_debt: u128,
): u128 {
    let precision = constants::rate_output_precision();
    let denominator =
        precision - self.get_col_from_ratios(col_per_debt);

    // Preserve existing behavior at equal or inverted decoded ratios.
    if (self.ref_ratio >= self.ratio) {
        let numerator =
            (self.debt - self.get_debt_from_ratios()) * precision;
        return numerator / denominator
    };

    let numerator =
        (self.debt as u256)
        * ((self.ratio - self.ref_ratio) as u256)
        * (precision as u256);

    let divisor =
        (self.ratio as u256) * (denominator as u256);

    (numerator / divisor) as u128
}
```

Customer Response:

Fixed in commit bced61.

Fix Review:

Fixed.

M-02: Partial branch ratio falls below the next perfect tick and blocks liquidation

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: packages/vaults/sources/logic/liquidate.move	Status: Fixed	

Description:

When liquidation reaches the liquidation-threshold tick with debt budget remaining, `end_liquidate()` stores the residual debt as a branch at `ref_tick + 1` with `partials = 1`. An occupied perfect tick at the threshold remains untouched. A later liquidation orders that branch ahead of the perfect tick by tick number, assuming the branch's decoded ratio is at least the perfect tick's ratio.

That assumption can fail. The branch decoder derives the bottom of its interval by dividing the already rounded upper-tick ratio; the perfect tick's ratio is calculated independently:

```
let ratio_one_less = math::mul_div_u128(
    ratio, constants::four_decimals(), tick_math::tick_spacing(),
);
let current_ratio = ratio_one_less + math::mul_div_u128(
    ratio - ratio_one_less,
    minima_tick_partials as u128,
    constants::x30(),
);
```

Consider a market with nine-decimal collateral, six-decimal debt, and `CF = 800`, `LT = 810`, `LML = 900`. At a price of `0.003` DEBT per COL, one position supplies 50,000,000 COL and borrows 108,000 DEBT at tick `-4094`; another supplies 319,018 COL and borrows 700 DEBT at tick `-4084`. A decline to `0.002669998` makes only the higher position liquidatable. Liquidating it to LT leaves a branch at tick `-4093`, `partials = 1`, holding `645520771095` raw debt shares. A further decline to `0.002654038` makes both that branch and the untouched tick `-4094` eligible for ordinary liquidation, without making either absorbable.

The branch now decodes below its lower perfect-tick reference:

```
Ratio at tick −4093                              609658433635
Estimated lower = floor(ratio × 10000/10015) 608745315661
Contribution from partials = 1                floor(913117974/1073741823) = 0
Decoded branch ratio                           608745315661
Perfect ratio at tick −4094                    608745315662
```

`get_debt_from_ratios()` therefore projects `floor(645520771095 × 608745315662 / 608745315661) = 645520771096` shares which is one more than the branch holds. `get_debt_liquidated()` attempts to subtract that amount from the branch debt and aborts before reaching the 108,000-DEBT position. A 100-DEBT liquidation of that position succeeds at the same price when the branch is absent.

An attacker controlling both positions can create the smaller branch after the first price move and delay liquidation of their larger unsafe loan after the second. The blockage lasts while this branch and reference remain in the affected ordinary-liquidation range. Repaying enough branch debt, removing the lower tick's debt, or a further price decline that makes the branch absorbable can restore progress; until then, the delay increases bad-debt exposure.

Recommendations:

Keep the existing partial interpolation, but clamp its decoded ratio to `get_ratio_at_tick(minima_tick - 1)` whenever the branch tick has a valid predecessor. Also, use this same bounded decoder for current and reference branches in liquidation, absorption, and position settlement, which currently repeats the interpolation separately.

Customer Response:

Fixed in commit c2589c & 8f91be.

Fix Review:

Fixed - *"The one-unit guard prevents the reported underflow for branch ticks through +11,610. Larger inversions remain possible above that range and are explicitly acknowledged under the documented vault-listing policy."*

Low Severity Issues

L-01: Fully liquidated positions cannot be closed without new funded activity

Severity: Low	Impact: Low	Likelihood: Low
Files: packages/vaults/sources/module/user.move	Status: Acknowledged	

Description:

Liquidation updates tick and branch accounting without updating each individual `Position`. As a result, a fully liquidated position can have no remaining collateral or debt while its stored fields still describe a borrowing position.

`close_position` checks only those stored fields:

```
self.is_supply_only_position && self.supply_amount == 0
```

The check therefore rejects the position even though liquidation has already cleared it.

Although `operate` can reconstruct the empty state, it rejects operations with no effective collateral or debt change before saving that state. Passing `Max` for withdrawal and repayment does not help: both resolve to zero after a complete liquidation.

The owner cannot close an economically empty position without introducing new funds. A deposit followed by withdrawal can update the stored state and allow closure, but this requires unnecessary funding and additional execution costs. It also makes cleanup depend on normal Liquidity deposit and withdrawal restrictions. Until this workaround succeeds, the owner cannot delete the position and its ownership capability through the close function. This also prevents recovery of the storage rebate associated with deleting these objects until closure succeeds.

Recommendations:

Update `close_position` to resolve the position's current state using the existing settlement logic before checking whether it is empty. Allow mutable vault access so any required accounting adjustments can be applied.

If the resolved collateral and debt are both zero, delete the position and its ownership capability without requiring a deposit or repayment. Otherwise, reject closure.

Customer Response:

Acknowledged - *"Confirmed: liquidation updates tick and branch accounting without touching individual Position objects, so a fully liquidated position keeps stale stored fields and position::delete (is_supply_only_position && supply_amount == 0) rejects it, while operate aborts with vault_invalid_operate_amount when both resolved amounts are zero (a Max withdraw/payback resolves to zero). This does not affect solvency or funds: the position holds nothing, and the only cost is a delayed storage-rebate recovery. The owner can close it by depositing at least the minimum operate amount and withdrawing it back (both in one PTB, so the deposit is returned and only gas and rounding are spent), which resyncs the stored state. Solana has the same close rule (verify_close_position) and zero-amount operate check, and EVM has the same zero-amount check, so the logic is left unchanged to avoid a Sui-only divergence."*

L-02: Withdrawal-gap check floors a share conversion that Liquidity ceils, so one raw unit of the buffer can be withdrawn

Severity: Low	Impact: Low	Likelihood: Low
Files: packages/vaults/sources/utils/operate.move	Status: Acknowledged	

Description:

`withdraw_gap` haircuts the vault's Liquidity supply before comparing it with Liquidity's withdrawal limit. The remainder is the extra buffer that ordinary `operate` withdrawals are not allowed to take, so liquidations still have room to pull collateral.

When the user is withdrawing, that check first converts the native token amount into Liquidity supply shares:

```
let user_withdrawal = math::mul_div_u128(
    withdraw_token_amount,
    constants::exchange_prices_precision(),
    liquidity_ex_price,
);
```

`mul_div_u128` floors. Immediately afterwards, `liquidity::supply` withdraws the same native amount from a with-interest position, and Liquidity converts it with `mul_div_ceil_u128`:

```
new_supply_interest_raw = math::mul_div_ceil_u128(
    amount as u128,
    constants::exchange_prices_precision(),
    supply_exchange_price,
);
```

Whenever `withdraw_token_amount × 1e12` is not divisible by the supply exchange price, the pre-check counts `N` shares and Liquidity burns `N + 1`. A withdrawal whose floored size equals the remaining headroom therefore passes `headroom >= user_withdrawal` and then consumes one raw share of the reserved gap.

The impact is that this rounding is not in favor of the protocol: the withdrawal buffer that `withdraw_gap` is meant to hold for liquidations can be exceeded by one Liquidity supply share per such withdrawal.

Recommendations:

Convert the requested withdrawal with the same ceiling Liquidity will use to burn shares:

```
let user_withdrawal = math::mul_div_ceil_u128(
    withdraw_token_amount,
    constants::exchange_prices_precision(),
    liquidity_ex_price,
);
```

Customer Response:

Acknowledged - *"The floor in check_if_withdrawal_safe_for_withdrawal_gap does differ from Liquidity's ceil when burning shares, so a withdrawal sized exactly to the headroom can exceed the gap by one raw supply share. This does not affect solvency: the gap is a soft buffer that keeps room for liquidations, Liquidity's hard withdrawal limit is still enforced in full, and the burn rounds against the withdrawer. The magnitude is about 1e-12 of a token unit and cannot be compounded. Matches the Solana reference (safe_div on the same conversion), so the logic is left unchanged."*

L-03: Zero-collateral liquidation reconstruction blocks collateral-only recovery

Severity: Low	Impact: Low	Likelihood: Low
Files: packages/vaults/sources/utils/operate.move	Status: Acknowledged	

Description:

A partially liquidated position can retain debt while its reconstructed collateral rounds down to zero. This occurs in `fetch_latest_position`, where collateral is calculated using floor division:

```
position_raw_col = math::mul_div_u128(
    position_raw_debt,
    tick_math::zero_tick_scaled_ratio(),
    final_ratio,
);
```

The position remains active when its debt exceeds the liquidation cleanup threshold. If the owner subsequently attempts to restore the position by depositing collateral without changing the debt, `operate` calls:

```
let old_ratio = old_state.get_old_ratio();
```

which calculates:

```
math::mul_div_u128(
    self.old_net_debt_raw,
    tick_math::zero_tick_scaled_ratio(),
    self.old_col_raw,
)
```

Because `old_col_raw` is zero, the transaction aborts with a division-by-zero error before the new collateral is persisted.

For example, consider nine-decimal collateral and debt, equal vault exchange prices, a collateral price of `3,000` debt tokens, and an `810` liquidation threshold. A partially liquidated position may retain `0.0000022` debt while its corresponding collateral is approximately:

```
0.0000022 / (3,000 * 810 / 1,000)
≈ 0.00000000090535
```

This rounds down to zero collateral, while the debt remains above the `0.000002` liquidation cleanup threshold. A collateral-only deposit then always reverts when calculating the old ratio.

As a result, the owner cannot restore the position using collateral alone and must additionally obtain and repay the debt token. This may prevent timely recovery and leave the position exposed to further liquidation, even though adding collateral strictly improves its health.

Recommendations:

Skip the old-ratio comparison when the reconstructed old collateral is zero:

```
public fun check_if_ratio_safe_for_deposit(
    memory_vars: &OperateMemoryVars,
    old_state: &OldState,
    new_net_debt_raw: u128,
) {
    if (old_state.old_col_raw() == 0) {
        return
    };

    let new_ratio = memory_vars.get_new_ratio(new_net_debt_raw);
    let old_ratio = old_state.get_old_ratio();

    assert!(
        !(new_ratio > old_ratio),
        errors::vault_invalid_payback_or_deposit(),
    );
}
```

Positive debt over zero collateral represents an infinite ratio, so adding positive collateral cannot worsen it. The subsequent position-safety check still rejects any deposit that does not restore the position below the liquidation threshold.

Customer Response:

Acknowledged - *"The behaviour is confirmed: a partially liquidated position whose reconstructed collateral floors to zero reverts on a collateral-only deposit, because get_old_ratio divides by old_col_raw. This does not affect solvency: it only arises when residual debt is within a few raw units of the liquidation cleanup threshold (dust), the abort happens before any state change, and the position stays fully liquidatable throughout. The owner can still recover by depositing collateral together with a 1-raw-unit payback (or a dust borrow), which skips the check_if_ratio_safe_for_deposit branch, so the debt-token cost is negligible. The logic is left unchanged to avoid adding a special case for dust."*

L-04: Combined repayment and withdrawal reverts above CF despite reducing risk

Severity: Low	Impact: Low	Likelihood: Low
Files: packages/vaults/sources/utils/operate.move	Status: Fixed	

Description:

A borrower whose position has become unsafe may repay debt while withdrawing some collateral in the same operation. If the repayment reduces the position's debt-to-collateral ratio, this is an atomic deleverage: the position becomes less risky despite the withdrawal.

Sui applies the CF limit to every withdrawal with debt remaining, regardless of the simultaneous repayment or the improvement in the final tick:

```
if (new_col.get_scaled_is_negative() || new_debt.get_scaled_is_positive()) {
    (exchange_rate_operate, vault.collateral_factor() as u128,
     errors::vault_position_above_cf(),)
} else {
    (exchange_rate_liquidate, vault.liquidation_threshold() as u128,
     errors::vault_position_above_liquidation_threshold(),)
};

// The resulting tick must satisfy the selected limit.
assert!(!tick.gt(limit_tick), error_code);
```

For example, with `CF = 800` and `LT = 900`, a price move could leave a position at a ratio of `920`. The borrower then repays debt and withdraws collateral in amounts that lower the ratio to `850`. The final position is below LT and no longer liquidatable, but the operation reverts because `850` remains above CF. A borrower who cannot fund a larger repayment is prevented from executing this risk-reducing unwind in one operation.

EVM permits this case. Its CF check is triggered when the new tick has not improved, or when net debt has increased beyond a small rounding allowance:

```
o_.oldTick <= o_.tick ||
(o_.debtRaw - o_.dustDebtRaw) >
    (((o_.oldNetDebtRaw * 1000000001) / 1000000000) + 1)
```

Thus, a withdrawal paired with repayment can proceed above CF when it lowers the tick without materially increasing net debt. Sui and Solana omit that exception.

Cross-reference: This CF/deleverage case is, also, covered within L-09 of Certora's Fluid Oracles audit report.

Recommendations:

For an operation that both withdraws collateral and repays debt, permit the result above CF when the new tick is strictly lower than the old tick and net debt has not increased beyond the existing rounding allowance.

Customer Response:

Fixed in commit 95ac3e.

Fix Review:

Fixed.

L-05: Unfunded borrow fee reduces depositor yield

Severity: Low	Impact: Low	Likelihood: Medium
Files: packages/vaults/sources/logic/operate.move	Status: Acknowledged	

Description:

The vault borrow fee is added only to the vault's internal debt. It is not borrowed from Liquidity, so the vault's recorded debt is larger than Liquidity's debt, and that gap grows with interest.

On a borrow, the fee is added to `debt_raw` only. `new_debt`, the amount actually taken from Liquidity, stays at the principal. The code expects a later rebalance to collect the fee.

```
let borrow_amount_with_fee: u128 =
    borrow_amount + math::mul_div_ceil_u128(
        borrow_amount,
        exchange_prices.borrow_fee as u128,
        constants::borrow_fee_precision(),
    );
// Only adding fee in debt_raw and not in new_debt as new_debt is debt that needs
to be borrowed from Liquidity
info.debt_raw = info.debt_raw + borrow_amount_with_fee;
info.new_debt
```

Liquidity's supply rate is computed from the tokens it actually lent. `calculate_exchange_prices` builds that rate from `total_borrow_with_interest`, `last_utilization`, and the borrow rate. The unfunded fee is in none of those.

```
// supply rate = borrowRate * ratioSupplyYield * (1e4 - fee).
let supply_rate = borrow_rate * ratio_supply_yield * (four_decimals - (self.fee
as u128));
```

`ratio_supply_yield` is scaled by `last_utilization`, which is the real borrow divided by the real supply. A smaller borrow means lower utilization, a lower supply rate, and slower growth of the deposit exchange price.

A 100 borrow with a 1 fee is 101 inside the vault and 100 at Liquidity. With 1,000 supplied, utilization is 100 / 1,000 = 10%. If the fee had been borrowed it would be 101 / 1,000 = 10.1%. Depositors earn the 10% rate.

After 10% interest the vault debt is 111.1 and the Liquidity debt is 110. The 1.1 is owed by the borrower and is not part of Liquidity's borrow, so it never enters the supply exchange price. Depositors do not receive that yield. A later rebalance can borrow the missing fee from that moment on, but it does not restore the yield missed before then.

The chain is: unfunded fee, lower Liquidity borrow, lower utilization, lower supply rate, slower deposit-price growth. The extra debt and its interest stay with the protocol.

Recommendations:

Borrow the fee from Liquidity in the same transaction as the principal, and send those tokens to the fee recipient. Utilization and the supply rate then include the fee immediately, and the interest on it is paid to depositors instead of remaining outside Liquidity.

Customer Response:

Acknowledged - *"This is intended design, inherited from Fluid VaultT1. The borrow fee is added to the vault's debt_raw only, and the permissioned rebalance later borrows the difference from Liquidity and sends it to the fee recipient. The fee is part of the borrower's debt from the moment it is charged, and the borrower pays it in full, including its interest. The vault's recorded debt is always at least Liquidity's, so solvency is not affected."*

L-06: Missing dust-debt reconciliation after full liquidation understates total_borrow

Severity: Low	Impact: Low	Likelihood: High
Files: packages/vaults/sources/logic/liquidate.move	Status: Acknowledged	

Description:

When a position is placed on a tick, its recorded debt includes placement dust introduced by tick rounding. The position and tick therefore store gross debt, while the vault's `total_borrow` includes only the net debt (`gross debt - placement dust`).

Liquidation operates on the gross debt recorded on the tick and branch, and settlement subtracts the liquidated amount from `total_borrow`. Once a position is fully liquidated, this subtraction also removes its placement dust, even though that dust was never included in `total_borrow`. The missing step is adding the realized dust back to the vault's total borrow.

For example, a position with net debt of 9,950 and placement dust of 50 contributes 10,000 to its tick but only 9,950 to `total_borrow`. Fully settling that gross debt subtracts 10,000, leaving `total_borrow` understated by 50. When the user subsequently calls `operate`, the fully liquidated position's dust is recorded in `absorbed_dust_debt`, but no reconciliation path consumes this amount and restores it to `total_borrow`.

Consequently, successful liquidations leave the vault's recorded debt below the debt attributable to its remaining positions. Rebalancing does not correct the discrepancy because the gross repayment also reduces the debt held in Liquidity.

As fully liquidated positions accumulate, the unreconciled dust progressively understates `total_borrow`, causing it to diverge from outstanding position debt and potentially making subsequent liquidations revert when the debt being settled exceeds the recorded total.

Recommendations:

Implement a reconciliation mechanism that adds realized placement dust back to `total_borrow` exactly once after full liquidation, then clears or marks the reconciled amount. This restores the vault's debt accounting and allows rebalancing to recognize the corresponding difference from Liquidity.

Track this separately from the remaining net debt recorded by the `debt < 2 × minimum debt` path. That path leaves debt on the branch and in `total_borrow`; adding its counter value back would double-count debt.

Customer Response:

Acknowledged - *"The EVM absorbDustDebt logic will be ported when it is implemented on Sui."*

L-07: A position accepted at the LT limit tick can be liquidated at the same price

Severity: Low	Impact: Medium	Likelihood: Low
Files: packages/vaults/sources/utils/liquidate.move packages/vaults/sources/utils/operate.move	Status: Acknowledged	

Description:

For a collateral deposit or debt repayment, `check_if_position_safe()` permits the resulting position when its tick is at or below the LT limit tick. Liquidation processes positions above its own LT limit tick. Both paths use the liquidation oracle price for these operations, but round the threshold calculation in a different order:

```
// check_if_position_safe(): apply LT before converting to Q48
let check_ratio = exchange_rate * threshold / constants::three_decimals();
let check_ratio = math::mul_div_u128(
    check_ratio,
    tick_math::zero_tick_scaled_ratio(),
    constants::rate_output_precision(),
);

// get_ticks_from_oracle_price(): convert to Q48 before applying LT
let liquidation_ratio = math::mul_div_u128(
    debt_per_col,
    tick_math::zero_tick_scaled_ratio(),
    constants::rate_output_precision(),
);
let threshold_ratio = math::mul_div_u128(
    liquidation_ratio,
    vault.liquidation_threshold() as u128,
    constants::three_decimals(),
);
```

Let the effective price be `P = 1_600_000_000_000_000` in 10^{15} precision, `LT = 625/1000`, and `Q` = 2^{48}. The operation calculation produces `Q`, or tick `0`; liquidation produces `Q - 1`, or tick `-1`.

Consider a market with `CF = 600`, `LT = 625`, and `LML = 700` and `liquidation_penalty = 1,000` (10%). A position opens at a price of `1.67` DEBT per COL with `100,000 COL9` supplied and `99,900 DEBT6` borrowed. At a price of `1.60000000`, the owner deposits another `1 COL9`. The deposit passes the LT check and the position remains at tick `0`. With no subsequent price or position change, liquidation treats tick `0` as above its tick-`-1` limit. A liquidator can repay `477.711784 DEBT6` and receive `328.426851117 COL9`, worth `525.4829617872 DEBT` at the oracle price. The borrower incurs a liquidation penalty immediately after an operation that classified the position's tick as within LT.

The same rounding order is used to calculate `max_tick`. With `CF = 550`, `LT = 600`, and `LML = 625`, it can make a tick-`0` position whose debt is below its LML allowance fully absorbable; collateral in the absorbed buffer can then be purchased below its oracle value.

This discrepancy occurs only near a tick boundary. In the LT example, increasing the quote to `1.60000001` makes both paths select tick `0`, ending the mismatch for future calls.

Recommendations:

In `get_ticks_from_oracle_price()`, calculate each LT and LML Q48 ratio from the already adjusted effective price with one division: `floor(price × threshold × 2^48 / (1000 × 10^15))`. Keep the existing limit tick as a baseline, and raise it only if the full-precision ratio selects a higher valid tick whose `get_ratio_at_tick()` value does not exceed that ratio. Leave price adjustment and `col_per_debt` unchanged.

Customer Response:

Acknowledged - *"The operate-side half does not exist on Sui: since M-01, deposits and paybacks skip the health check and never read the oracle, and the remaining check_if_position_safe only enforces the collateral factor, so no position is "accepted at the LT limit tick" by an operation (EVM likewise checks only CF on withdraw/borrow; only Solana has a separate LT check for deposit/payback). The liquidation side (get_ticks_from_oracle_price) scales the price to Q48 before applying LT/LML, which floors liquidation_tick and max_tick by at most one tick against the exact ratio (reproduced: P = 1.6e15, LT = 625 gives ratio Q-1, tick -1, instead of Q, tick 0). This is identical in EVM and Solana, only matters when the price sits exactly*

on a tick boundary, and resolves itself on the next price change. The liquidation math is left unchanged to stay aligned with the references."

L-08: Over-debiting one position can clear a shared tick and block another borrower's exit

Severity: Low	Impact: Low	Likelihood: Low
Files: packages/vaults/sources/utils/operate.move	Status: Acknowledged	

Description:

Placement adds a position's gross debt to `Tick.raw_debt` using the ratio returned by the inverse tick lookup. On a later operation, `get_position_info()` reconstructs that debt using the stored tick's forward ratio, with additional rounding. The reconstructed amount can exceed what placement credited. Ordinary settlement subtracts it from the **shared** tick and clears the tick when the remainder falls below its minimum:

```
let mut remaining_debt =
    if (tick_raw_debt > memory_vars.debt_raw()) {
        tick_raw_debt - memory_vars.debt_raw()
    } else { 0 };

if (remaining_debt < constants::get_minimum_tick_debt(debt_decimals)) {
    remaining_debt = 0;
    tick::update_tick_has_debt(vault.tick_bitmap_mut(), position_tick, false);
};
```

For example, two positions share tick `-16379`. Placement uses Q48 ratio `6129`, while reconstruction uses `6130`. With `267,119,752,898,412,544` raw collateral units, the first position contributes `5,816,421` gross debt shares but is later reconstructed as `5,817,371`. The second position contributes `1,002` shares, making the tick total `5,817,423`.

When the first borrower fully exits, settlement leaves `5,817,423 - 5,817,371 = 52` shares. That is below the `100`-share tick minimum, so the tick is cleared while the second position remains open. The second borrower's next operation aborts at `vault_tick_is_empty`, blocking repayment and collateral withdrawal.

A new borrow at the same tick can reactivate it and allow the second borrower to exit. That does not repair the accounting: the new borrower can then become the stranded position.

Recommendations:

Make the gross debt credited in `compute_placement()` exactly reproducible from the position's stored tick and collateral, and use that same amount for ordinary settlement and dust accounting.

Customer Response:

Acknowledged - *"Only dust positions are affected and they are not locked; the exit unblocks as soon as someone creates a position at that tick. This is known behavior, documented in a code comment on EVM."*

L-09: Deposit and partial repay require an operate-grade oracle quote even though the health check only uses the liquidate rate

Severity: Low	Impact: Low	Likelihood: Low
Files: packages/vaults/sources/logic/operate.move	Status: Fixed	

Description:

`check_if_position_safe` always reads both oracle rates, then picks one. Withdraw and borrow use the operate rate and collateral factor. Deposit and payback use the liquidate rate and liquidation threshold.

On Sui, reading `rate_operate` aborts if that field is missing. A dual quote cannot be built unless the feed also passes operate checks (600s freshness, tighter confidence). A liquidate-only quote (7200s, looser confidence) is valid for liquidation but cannot be used here.

The result: when the feed is still good for liquidation but not for operate, a user cannot deposit collateral or partially repay to get back under the liquidation threshold. Liquidators can still liquidate with the liquidate-only quote. Full repayment still works because it skips the health check.

Cross-reference: This quote-availability case is, also, covered within L-10 of Certora's Fluid Oracles audit report.

Recommendations:

Decide the rate first, then fetch only that rate.

- Withdraw or borrow: operate rate + collateral factor.
- Deposit or payback: liquidate rate + liquidation threshold (`get_exchange_rate_liquidate` / a liquidate-only quote).

Do not call `get_both_exchange_rate` or `rate_operate` on the deposit/payback path.

Customer Response:

Fixed in commit 95ac3e.

Fix Review:

Fixed.

Informational Severity Issues

I-01: Cap-free deposit re-fills supply_amount, griefing the empty-position gate in position::delete

Files:

packages/vaults/sources/module/user.move

Description:

`Position` is a shared object. Deposit and payback do not require `PositionOwnerCap`. Close requires the cap and an empty book: `is_supply_only_position && supply_amount == 0`.

Anyone can deposit the protocol minimum (`min_operate` = 1000 native units when decimals ≥ 4) onto a position the owner just emptied. That writes a non-zero `supply_amount`, so `close_position` aborts with `vault_position_not_empty`. The owner must spend another operate (withdraw) plus close. The attacker can repeat it.

Recommendations:

Require `PositionOwnerCap` for `operate` calls when `supply_amount == 0`, or document that owners must call `max_withdraw` and `close_position` in the same PTB.

Customer Response:

Acknowledged - *"Confirmed: Position is shared, operate only needs the owner cap for withdraw/borrow, and position::delete requires is_supply_only_position && supply_amount == 0, so a third-party minimum deposit onto an emptied position makes close_position abort with vault_position_not_empty. Cap-free deposits are intended (anyone can top up or repay for any position). This does not affect solvency or funds: the attacker's deposit lands in the victim's position, so each attempt costs the attacker more than the victim's one extra withdraw. Owners avoid the race entirely by calling max_withdraw and close_position in the same PTB, since a shared object cannot be modified between commands of one transaction. The logic is left unchanged rather than gating deposits on the cap."*

I-02: LML validation rejects inputs that become valid after truncation

Files:

packages/vaults/sources/state/risk_config.move

Description:

In `risk_config::new()`, `liquidation_max_limit + liquidation_penalty <= 9970` is checked before dividing LML by 10 for storage. Consequently, discarded digits can cause rejection even when the effective configuration satisfies the limit. This affects `update_liquidation_max_limit`, and the construction of `RiskConfig` for `update_core_settings` and `init_vault_config`.

```
check_liquidation_max_limit_and_penalty(
    liquidation_max_limit, liquidation_penalty,
);

let collateral_factor = collateral_factor / 10;
let liquidation_threshold = liquidation_threshold / 10;
let liquidation_max_limit = liquidation_max_limit / 10;
```

For example, `LML = 9001` and `penalty = 970` are rejected because their sum is `9971`. However, LML would be stored as `900`, equivalent to `9000` at the penalty's scale, making the effective sum `9970`.

Recommendations:

Validate using `(liquidation_max_limit / 10) * 10` so the check reflects the stored LML while preserving the penalty's scale.

Customer Response:

Acknowledged - *"new() checks the raw 1e2 sum before the /10 truncation, so an off-grid input (e.g. LML 9001 + penalty 970) is rejected even though it would store as a valid 900. The check only rejects safe configs, never admits unsafe ones, and an explicit revert on admin input beats silently rounding it down. Matches the Solana reference; the admin resubmits a multiple of 10."*

I-03: Vault configuration events omit the affected vault and use inconsistent risk-parameter scales

Files:

packages/vaults/sources/module/admin.move
packages/vaults/sources/module/user.move

Description:

`mint_and_install_caps` installs the collateral cap, debt cap, or both without emitting a Vault event. In addition, `LogUpdateSupplyRateMagnifier`, `LogUpdateBorrowRateMagnifier`, `LogUpdateCollateralFactor`, `LogUpdateLiquidationThreshold`, `LogUpdateLiquidationMaxLimit`, `LogUpdateWithdrawGap`, `LogUpdateLiquidationPenalty`, `LogUpdateBorrowFee`, `LogUpdateCoreSettings`, `LogUpdateOracle`, and `LogUpdateExchangePrices` do not include the affected vault ID. The individual collateral factor, liquidation threshold, liquidation max limit, and withdrawal gap events also emit the original 1e4 input, whereas `LogUpdateCoreSettings` emits the truncated 1e3 values stored in `RiskConfig`.

```
vault.add_caps(col_cap, debt_cap); // No Vault event

events::emit_log_update_liquidation_threshold(
    liquidation_threshold, // Original 1e4 input
);

events::emit_log_update_core_settings(
    // ...
    risk.liquidation_threshold(), // Stored 1e3 value
    // ...
);

public struct LogUpdateExchangePrices has copy, drop {
    vault_supply_exchange_price: u128,
    vault_borrow_exchange_price: u128,
    liquidity_supply_exchange_price: u128,
    liquidity_borrow_exchange_price: u128,
    // No vault ID
}
```

For example, `update_liquidation_threshold(8109)` stores `810` and emits `8109`, while setting the same effective value through `update_core_settings` emits `810`. When several vaults exist, the listed update events also cannot be associated with a particular vault using their event fields alone. Liquidity emits `LogMintUserCap`, but the Vault event history does not indicate which cap or caps were installed into the market.

Recommendations:

Emit a cap-installation event, add the vault ID to every market-specific update event, and consistently emit the stored risk values in 1e3 precision.

Customer Response:

Fixed in commit fea7f55 - *"(1) No Vault event for mint_and_install_caps / add_caps: Liquidity already emits LogMintUserCap for each cap minted, and the holder is derived from the vault object ID, so indexers can join on it. (2) The individual LogUpdateCollateralFactor / LogUpdateLiquidationThreshold / LogUpdateLiquidationMaxLimit / LogUpdateWithdrawGap events still emit the 1e4 admin input, while LogUpdateCoreSettings emits the stored 1e3 values; the input scale is what the admin passed in, and the vault ID now lets indexers tell which vault each event belongs to."*

Fix Review:

Partially fixed.

I-04: Incomplete initialization validation permits unusable vault markets

Files:

packages/vaults/sources/module/admin.move

Description:

`init_vault_config` creates and registers a `VaultMarket` without enforcing two prerequisites required by its Liquidity integration:

- Collateral and debt must be different token types. Configuration and state initialization can succeed when both types are identical, because state initialization only needs immutable reserve references. However, `mint_and_install_caps` attempts to create two Liquidity capabilities using the same vault holder and token type. Liquidity permits only one capability for that combination, so installing both capabilities fails. Installing them in separate transactions does not resolve the conflict.
- Both tokens must have between 2 and 9 decimals. Sui permits coins with 0 or 1 decimal places, and vault configuration accepts these coins because its decimal validation only enforces the upper bound of 9. Liquidity reserve creation, however, requires at least 2 decimals. Consequently, the required reserve cannot be created, preventing completion of vault state initialization.

When configuration creation is committed separately from the remaining setup steps, these inputs leave a registered shared vault that cannot complete initialization. The impact is failed deployments, unnecessary setup costs, and unusable vault objects that require creating a correctly configured replacement.

Recommendations:

Enforce both prerequisites in `vault_market::create`, before allocating the vault ID and constructing the shared market:

1. Reject identical collateral and debt types by comparing `std::type_name::with_defining_ids<Col>()` and `std::type_name::with_defining_ids<Debt>()`.
2. Require both decimal counts to fall within Liquidity's supported 2–9 range.

Use explicit configuration errors so unsupported markets are rejected at creation rather than failing during subsequent setup.

Customer Response:

Fixed in commit fea7f55.

Fix Review:

Fixed.

I-05: Withdraw gap skipped when liquidity limit is 0 in check_if_withdrawal_safe_for_withdrawal_gap

Files:

packages/vaults/sources/utils/operate.move

Description:

`check_if_withdrawal_safe_for_withdrawal_gap` is meant to keep a vault-owned buffer of the vault's supply at liquidity: at most `supply * (1000 - withdraw_gap) / 1000` can leave in one operate (stored gap is 1e3, so `100` = 10%).

It returns immediately if `withdraw_gap == 0` or if `liquidity_withdrawal_limit == 0`.

That second return mixes up two different limits. Liquidity treats a limit of `0` as "full withdraw allowed." It is not "the vault asked for a zero buffer." The vault can still have a non-zero `withdraw_gap`.

On a market whose liquidity supply position has never activated the expand-limit (typical: small vault, or `last_limit` still 0), operate never runs the gap math. One collateral withdraw can pull the vault's liquidity supply down to zero even though governance set a gap.

Recommendations:

Only skip the gap check when `withdraw_gap` is 0, not when the liquidity limit is 0. If the liquidity limit is 0, still enforce the gap against the vault's supply at liquidity.

Customer Response:

Acknowledged - *"Matches the EVM and Solana references, where a limit of 0 means Liquidity has no active withdrawal limit and the gap, a haircut on that limit, is not applied. Only happens on markets whose supply position never activated its expand-limit, which production configuration avoids; no fund-loss path."*

I-06: Magnifier rate_change floors on negative supply and positive borrow, underpaying the rebalancer by 1 wei

Files:

packages/vaults/sources/math/exchange_price.move

Description:

After Liquidity growth is applied, a non-zero rate magnifier adjusts the vault exchange price by

```
fun rate_change(old_price: u128, time_diff: u128, magnitude: u16): u128 {
    (((old_price as u256) * (time_diff as u256) * (magnitude as u256)
        / (constants::four_decimals() as u256)
        / (constants::seconds_per_year() as u256)) as u128)
}
```

That value is always rounded down, then subtracted from the supply exchange price when the supply magnifier is negative, or added to the borrow exchange price when the borrow magnifier is positive.

Flooring shrinks the delta by 1 exchange-price unit whenever `old_price * dt * mag` is not divisible by `1e4 * seconds_per_year`. A smaller subtract leaves the supply exchange price 1 unit too high, so `col_recorded` is 1 unit too high and the withdraw gap is 1 native wei too small. A smaller add leaves the borrow exchange price 1 unit too low, so `debt_recorded` is 1 unit too low and the borrow gap is 1 native wei too small. `rebalance_with_amounts` settles those preview amounts. The impact is that this rounding is not in favor of the protocol: the rebalancer takes 1 wei less on the withdraw or borrow.

Recommendations:

Round `rate_change` up when the delta is subtracted from the supply exchange price or added to the borrow exchange price:

```
fun rate_change(old_price: u128, time_diff: u128, magnitude: u16, round_up:
bool): u128 {
    let n = (old_price as u256) * (time_diff as u256) * (magnitude as u256);
    let d = (constants::four_decimals() as u256) * (constants::seconds_per_year()
as u256);
    if (round_up) {
```

```
        (((n + d - 1) / d) as u128)
    } else {
        ((n / d) as u128)
    }
}
```

Pass `round_up = true` for a negative supply magnifier and a positive borrow magnifier.

Customer Response:

Acknowledged - *"At most 1 wei per rebalance, no user-fund or solvency impact; floors the same way as the EVM/Solana references."*

I-07: LogUserPosition reports gross tick debt instead of the user's net borrow

Files:

packages/vaults/sources/logic/operate.move

Description:

After operate places a position, `LogUserPosition.borrow` is filled from `memory_vars.debt_raw()`. At that point `debt_raw` is the gross tick debt: the amount implied by rounding the ratio up to the next tick. The user's actual liability is `debt_raw - dust_debt_raw` (`get_net_debt_raw()`).

The same event also unscales those raw shares straight to token decimals. Raw shares live in 9-decimal space, so unscaling looks like a token amount even though it is still a share count, not tokens. Convert shares through the exchange price first (actual raw), then unscale to token decimals, and emit net — not gross — so `col` / `borrow` are real token amounts.

NOTE: `LogAbsorb` has the same issue: it unscales raw absorbed col/debt into token decimals with no exchange price, so `col_amount` / `debt_amount` look like tokens but are still vault shares (gross).

Recommendations:

Emit net debt as a token amount: `get_net_debt_raw()` × borrow exchange price, then `unscale_amounts` to token decimals. Apply the same convert-then-unscale path to `col`.

Emit `LogAbsorb` the same way.

Customer Response:

Acknowledged - *"Intentional: LogUserPosition and LogAbsorb follow the Solana reference (logic/operate.rs, logic/liquidate.rs), which unscales the raw share values directly with no exchange-price conversion, and borrow is the gross tick debt. The values are informational only and feed no accounting or safety logic; indexers can derive net token amounts from the vault exchange prices and the position's dust debt. EVM has no LogUserPosition (LogOperate carries the actual token deltas), and its LogAbsorb emits the raw values as well. The logic is left unchanged to keep event semantics aligned with Solana."*

I-08: Narrow LT–LML settings can eliminate ordinary liquidation

Files:

packages/vaults/sources/state/risk_config.move

Description:

`RiskConfig::new` accepts any configuration satisfying `CF < LT < LML`. Liquidation, however, converts LT and LML into discrete ticks. If both limits map to the same tick, there is no tick in the ordinary-liquidation band: positions above that tick are fully absorbed before ordinary liquidation runs. Thus valid risk settings can unintentionally make liquidation all-or-nothing.

For example, inputs `CF = 8_000`, `LT = 9_000`, and `LML = 9_010` are stored as `800`, `900`, and `901` in `1_000` precision. With an oracle rate of `877_100_000_000_000` and vault exchange prices of `1e12`, the calculation produces:

```
LT ratio  = 222_193_531_865_624 → tick -158
LML ratio = 222_440_413_567_697 → tick -158
```

A position at tick `-157` is therefore fully absorbed; it cannot undergo ordinary liquidation at this price despite the configured LT and LML being different.

Recommendations:

After converting the inputs to stored `1_000` precision, require `liquidation_max_limit >= liquidation_threshold + 2`: within the allowed parameter range, a two-unit gap exceeds one tick's `10_015 / 10_000` ratio step, including the threshold calculations' integer rounding.

Customer Response:

Fixed in commit fea7f55.

Fix Review:

Fixed.

I-09: emit_log_liquidation_rounding_diff event is emitted on a path that always aborts

Files:

packages/vaults/sources/utils/liquidate.move

Description:

When priced debt exceeds the liquidator's amount by more than `MAX_LIQUIDATION_ROUNDING_DIFF` (100), `get_actual_amounts` emits `LogLiquidationRoundingDiff` and then aborts (`vault_invalid_liquidation`).
On Sui an abort rolls back the transaction, so the event is never committed. Off-chain listeners that only index successful events never see `vault_id`, `actual_debt_amt`, `debt_amount`, or `diff`. Failed txs still expose the abort code only.

Recommendations:

Drop the emit on the abort path, or emit only if the tx succeeds. Do not rely on this event for monitoring on Sui. The abort code is the on-chain signal.

Customer Response:

Fixed in commit fea7f55.

Fix Review:

Fixed.

I-10: Unfunded borrow fee prevents full repayment

Files:

packages/vaults/sources/logic/operate.move

Description:

A borrow adds the fee only to the vault's internal debt. The amount borrowed from Liquidity stays at the principal. `get_new_position_info` records `borrow_amount + fee` in `debt_raw`, then returns the original `new_debt` for the Liquidity call. The comment says the fee is collected later by rebalance.

A max repayment prices the whole internal debt, including the fee:

```
let payback_amount: u128 = math::mul_div_ceil_u128(
    info.debt_raw,
    exchange_prices.borrow_ex_price,
    epp,
);

info.debt_raw = 0;

structs::scaled_negative(payback_amount)
```

That amount is paid to the vault's shared Liquidity borrow. Liquidity converts it back to raw and requires the position to cover it:

```
new_borrow_interest_raw =
    math::mul_div_u128(
        amount as u128,
        constants::exchange_prices_precision(),
        borrow_exchange_price,
    );
assert!(user_borrow_position_amount >= new_borrow_interest_raw,
lib_errors::overflow());
```

Liquidity's raw debt is only the principal, so the fee-inclusive payback is larger than the position and the transaction aborts.

Example: borrow 100 with a 1% fee. The vault records 101. Liquidity records 100. Max repayment tries to clear 101 and reverts.

Closing the gap requires `rebalance`, which only an authorized rebalancer can call. Until that rebalancer borrows the fee difference, the user cannot fully repay. If several users share the vault's Liquidity debt, one full repayment can consume another user's principal, and the last borrower is the one who gets stuck.

Recommendations:

Borrow the fee from Liquidity in the same transaction as the principal, and send those tokens to the fee recipient. The vault debt and the Liquidity debt then match, so a max repayment clears the position without a rebalancer.

If the fee must stay inside the vault, cap a max repayment at the vault's actual Liquidity debt and settle the fee internally. A user must be able to reach zero debt without a privileged rebalance.

Customer Response:

Acknowledged - *"Intended design, inherited from Fluid VaultT1: the borrow fee is added to debt_raw only, and the permissioned rebalance later borrows the difference from Liquidity and sends it to the fee recipient. This does not affect solvency: the vault's recorded debt is always at least Liquidity's, and the borrower pays the fee in full. The effect is liveness only: a max repayment can abort at Liquidity's user_borrow_position_amount >= new_borrow_interest_raw check until a rebalance runs, and the gap is bounded by the accrued fee. Partial paybacks are unaffected. Rebalancing on a regular cadence is an operational requirement. The logic is left unchanged to match the reference and avoid a Sui-only divergence."*

I-11: Redundant bitmap clear when the settled tick is the top tick

Files:

packages/vaults/sources/logic/operate.move

Description:

In `settle_existing_position` inside `operate_logic::operate`, a position on the current top tick whose remaining debt falls below the minimum calls `VaultMarket::set_top_tick`. That function clears the old top tick's bitmap bit through `tick::fetch_next_top_tick`. `settle_existing_position` then calls `tick::update_tick_has_debt` on the same tick and clears the bit again. The second clear does not change the top tick, the bitmap, or accounting.

Recommendations:

In `settle_existing_position`, call `tick::update_tick_has_debt` only in the `else` branch, when the settled tick is not the top tick.

Customer Response:

Acknowledged - *"Confirmed: tick::fetch_next_top_tick clears the old top tick's bit in the bitmap before scanning, so the following tick::update_tick_has_debt(.., false) in settle_existing_position clears an already-cleared bit. The second clear is idempotent and changes no top tick, bitmap or accounting, so there is no solvency or functional impact; the cost is one extra table access on a rare path (a top-tick position settling below the minimum). In EVM the same two calls are not redundant, because _fetchNextTopTick only masks the top tick in memory and _updateTickHasDebt does the storage write, so the unconditional clear is kept to stay aligned with the reference and to remain correct if fetch_next_top_tick stops clearing."*

I-12: Relative liquidation minimum rounds down by one debt share

Files:

packages/vaults/sources/logic/liquidate.move

Description:

The liquidation entrypoint requires a request to meet both an absolute debt minimum and a minimum proportional to the vault's `total_borrow`. The proportional check rounds down before comparing it with the liquidator's requested debt-share budget:

```
assert!(
    total_debt / constants::billion() <= current_data.debt_remaining(),
    errors::vault_invalid_liquidation_amt(),
);
```

When `total_debt` is not divisible by `1_000_000_000`, this accepts a budget one raw debt share below one-billionth of the vault's debt. For example, with `total_debt = 1,000,000,001,001`, the exact threshold is `1,000.000001001` shares. A request for `1,000` shares passes, although meeting the threshold requires `1,001`.

Recommendations:

Round the proportional minimum up before comparing it with `debt_remaining`.

Customer Response:

Acknowledged - *"The floor division is confirmed: with total_debt = 1,000,000,001,001 a request for 1,000 shares passes although the exact threshold is 1,000.000001001. This does not affect solvency, funds, UX or operations: the check is an anti-dust guard (it ensures each liquidation moves at least one partial), not a solvency invariant, and the 1e9 bound is approximate by design (EVM notes partials precision is "slightly above 1e9"). The deviation is at most one raw debt share, about 1e-9 relative, and the liquidator still pays full price for the debt it liquidates. Matches the EVM reference (temp2_ / 1e9 > debtRemaining), so the logic is left unchanged to avoid a Sui-only divergence."*

I-13: LogInitVaultState does not include the initial pool exchange prices

Files:

packages/vaults/sources/module/admin.move

Description:

When a vault is initialized, it stores the current pool supply and borrow exchange prices as starting values. The vault's own exchange prices start at `1e12`, representing `1.0`.

However, the `LogInitVaultState` event only includes the vault ID. Without the initial pool prices, indexers relying only on events cannot reconstruct the vault's starting price basis or accurately track price growth from initialization.

Recommendations:

Include the initial pool supply and borrow exchange prices in the `LogInitVaultState` event.

Customer Response:

Acknowledged - *"The initial pool supply and borrow exchange prices are written to the VaultMarket object (liquidity_supply_exchange_price, liquidity_borrow_exchange_price) in the same transaction as init_vault_state, so they can be read off-chain at the checkpoint of the LogInitVaultState event, which carries the vault ID. The vault's own prices always start at the constant EXCHANGE_PRICES_PRECISION (1e12), and every later price change is covered by LogUpdateExchangePrices. The event is left unchanged."*

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline and is widely used by developers and security researchers during audits and bug bounties.

Certora also provides architectural design reviews, where researchers analyze system design and trust boundaries early to identify structural risks and reduce complexity before implementation.

Certora conducts off-chain audits covering backend services, APIs, frontend, and mobile applications, focusing on vulnerabilities in authentication, data handling, key management, and wallet-related workflows.

Certora also offers Penetration Testing and Red Teaming to simulate real-world attacks and uncover exploitable weaknesses across infrastructure, applications, and business logic.

In addition, Certora provides operational security (OpSec) assessments, reviewing key management, access controls, and operational processes to strengthen overall security posture.