code4rena

Jupiter Lend

Smart Contract Security Assessment

Audit dates: Feb 12 — Mar 13, 2026

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Jupiter Lend smart contract system. The audit took place from February 12 to March 13, 2026.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 3 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 86 QA reports compiling issues with a risk rating of LOW severity or informational.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Jupiter Lend team.

Following the audit, the Jupiter Lend team provided additional context to the judge regarding several Medium-severity findings, including details on design intent and operational safeguards that were not documented in the original contest materials. After reviewing this context, the judge determined that the findings in question warranted a Low severity rating for the purposes of this report. The severity levels in this document have been adjusted accordingly. As Code4rena reports do not list individual Low-severity findings in full, these findings are not detailed in this document but can be viewed, along with all other Low-severity findings from this audit, via the link provided in the Low Risk and Informational Issues section below.

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. It is recommended that a follow-up audit and development of a more complex stateful test suite be undertaken prior to continuing to deploy significant monetary capital to production.

Scope

The code under review can be found within the Jupiter Lend Solana Program repository, and is composed of 120 files written in the Rust programming language and includes 15,195 lines of Rust code.

The code in C4's Jupiter Lend repository was pulled from:

- Repository: https://github.com/Instadapp/fluid-solana-programs
- Commit hash: `626b177f235224bc5d074d39439cd2558f542886`

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/informational.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (3)

[M-01] Broken safety cap in `liquidate.rs` causes permanent DoS of liquidation engine during market crashes, leading to insolvency.

Submitted by ranzo, also found by 0xABSattar, AcsonBarbosa, edoscoba, happykilling, Khan2018, and legat

- `programs/vaults/src/utils/liquidate.rs` #L152-L154
- `programs/vaults/src/utils/operate.rs` #L41-L42

In `get_ticks_from_oracle_price()` (liquidate.rs) and `get_sanitized_exchange_rate()` (operate.rs), the protocol caps `debt_per_col` / `exchange_rate` at `10u128.pow(26)` (1e26) to prevent precision loss. After capping, the value is converted into a `ratioX48` format via:

```
let liquidation_ratio: u128 = safe_multiply_divide(
    debt_per_col,
    TickMath::ZERO_TICK_SCALED_RATIO, // 2^48 = 281474976710656
    10u128.pow(RATE_OUTPUT_DECIMALS), // 1e15
```

```
)?;
```

This ratio is then multiplied by the liquidation threshold (e.g., 900/1000 = 90%) and passed to `TickMath::get_tick_at_ratio()`, which enforces a hard upper bound of `MAX_RATIOX48 ≈ 1.30e25`.

The developer's own inline comment confirms they verified the math for `debt_per_col = 4e25` (which fits safely), but then set the cap at `1e26`—more than **2× the safe limit**—assuming the liquidation threshold multiplier would bring the result back within bounds. This assumption is incorrect: the threshold is applied **after** the conversion to `ratioX48`, so the intermediate `liquidation_ratio` already overflows `MAX_RATIOX48` before any scaling down occurs.

Root Cause

The cap value `1e26` was chosen based on an incorrect analysis. The developer's comment in `liquidate.rs` shows the safe boundary calculation was done for `raw_col_per_debt` (the inverse), not for `debt_per_col` used in the `liquidation_ratio` path. The two variables are computed from `debt_per_col` but feed into **different** downstream calculations with different bounds:

- `raw_col_per_debt = 1e30 / debt_per_col` → at `debt_per_col = 4e25`, this gives `~4e25 * 2^48 / 1e15 ≈ 1.13e25 < MAX_RATIOX48` ✓
- `liquidation_ratio = debt_per_col * 2^48 / 1e15` → at `debt_per_col = 1e26`, this gives `≈ 2.81e25 > MAX_RATIOX48` ✗

The cap protects one path but breaks the other.

Impact

- **Liquidation DoS:** When extreme market divergence pushes `debt_per_col` into the range `[~4.62e25, 1e26]`, calls to `get_ticks_from_oracle_price()` revert with `LibraryTickRatioOutOfBounds`. This blocks **all** liquidations for the affected vault precisely when liquidation is most critical (collateral collapse or debt spike scenarios).
- **Operate DoS:** The same cap in `get_sanitized_exchange_rate()` (operate.rs) means `check_if_position_safe()` also reverts, blocking deposits, withdrawals, borrows, and paybacks—freezing the vault entirely.
- **Bad debt accumulation:** Positions that should be liquidated cannot be, leading to unbounded protocol loss. The safety mechanism (cap) fails its stated purpose of keeping the math within bounds.
- **The cap was designed to enable graceful degradation under extreme prices.** Instead, it creates a hard revert window between `~4.62e25` and `1e26` where the protocol is completely inoperable.

Proof of Calculation

STEP	VALUE
`debt_per_col` (at cap)	`1e26` = `100_000_000_000_000_000_000_000_000`
`ZERO_TICK_SCALED_RATIO` (2^48)	`281_474_976_710_656`
`RATE_OUTPUT_DECIMALS`	15 → divisor = `1e15`
`liquidation_ratio`	`1e26 × 281474976710656 / 1e15 ≈ 2.81e25`
`threshold_ratio` (LT=90%)	`2.81e25 × 900 / 1000 ≈ 2.53e25`
`max_ratio` (LML=95%)	`2.81e25 × 950 / 1000 ≈ 2.67e25`
`MAX_RATIOX48`	`≈ 1.30e25`
Result	`2.53e25 > 1.30e25` → **REVERT**

Reverse Calculation for the Safe Cap

```
MAX_RATIOX48 = 1300208813309603656541429 5 ≈ 1.30e25
max safe liquidation_ratio = MAX_RATIOX48
max safe debt_per_col = MAX_RATIOX48 * 1e15 / 2^48
                      ≈ 1.30e25 * 1e15 / 2.81e14
                      ≈ 4.62e25
```

With a typical LT of 90%, the effective safe cap would be `~4.62e25 / 0.9 ≈ 5.13e25`. The current cap of `1e26` is nearly 2× too high.

Even a conservative LT of 50% yields `4.62e25 / 0.5 = 9.24e25`, still below `1e26`.

Recommended Mitigation Steps

Reduce the cap to align with `TickMath::MAX_RATIOX48`. A safe cap should account for the highest possible LT/LML multiplier. Since `liquidation_max_limit` can be close to 100% (e.g., 950/1000 = 95%), the cap must satisfy:

```
cap * 2^48 / 1e15 * (LML / 1000) ≤ MAX_RATIOX48
```

For `LML = 999` (near 100%, worst case):

```
cap ≤ MAX_RATIOX48 * 1e15 / 2^48 * 1000 / 999 ≈ 4.62e25
```

Apply the fix in both `liquidate.rs` and `operate.rs`:

```
  // liquidate.rs - get_ticks_from_oracle_price
- if debt_per_col > 10u128.pow(26) {
-     debt_per_col = 10u128.pow(26);
+ // Cap derived from MAX_RATIOX48 * 1e15 / 2^48 ≈ 4.62e25
+ // Rounded down for safety margin
+ if debt_per_col > 46_000_000_000_000_000_000_000_000_000 {
+     debt_per_col = 46_000_000_000_000_000_000_000_000_000;
  }
```

```
  // operate.rs - get_sanitized_exchange_rate
- if exchange_rate > 10u128.pow(26) {
-     exchange_rate = 10u128.pow(26);
+ if exchange_rate > 46_000_000_000_000_000_000_000_000_000 {
+     exchange_rate = 46_000_000_000_000_000_000_000_000_000;
  }
```

Proof of Concept

The vulnerability is a pure arithmetic overflow that can be demonstrated with a calculation trace. No on-chain PoC is needed as the revert path is deterministic.

Step-by-Step Scenario:

1. A vault is configured with `liquidation_threshold = 900` (90%) and `liquidation_max_limit = 950` (95%).
2. The collateral token's price collapses, or the debt token's price spikes, or vault exchange price magnifiers cause significant divergence between `supply_ex_price` and `borrow_ex_price`.
3. `debt_per_col` computes to a value in the range `[4.62e25, 1e26]`, for example `7e25`.
4. The cap at `1e26` does **not** activate (value is below cap).
5. `liquidation_ratio = 7e25 * 2^48 / 1e15 ≈ 1.97e25`.
6. `threshold_ratio = 1.97e25 * 900 / 1000 ≈ 1.77e25`.
7. `1.77e25 > MAX_RATIOX48 (1.30e25)` → `get_tick_at_ratio` reverts.
8. All liquidation calls fail. Bad debt accumulates.

Note: Even without reaching the cap, `debt_per_col` values above `~4.62e25` can cause the revert depending on the LT/LML configuration. The cap merely **fails to prevent** this scenario

—it should reject values above the safe threshold but currently allows values up to 2× the safe limit.

[M-02] Interest-bearing suppliers are over-credited when interest-free supply coexists, creating a reserve deficit

Submitted by qed

- `programs/liquidity/src/state/token_reserve.rs` #L524-L530
- `programs/liquidity/src/state/token_reserve.rs` #L449-L466

`calculate_exchange_prices()` applies an amplified supply rate multiplicatively to the current `supply_exchange_price`, but the amplification factor is derived from a conservation relationship that only holds at initialization (`Ps == 1e12`). Once `Ps` has increased, repeated accrual credits more value to interest-bearing suppliers than borrowers fund. The growing deficit means early redeemers withdraw inflated value while later redeemers face a shortfall.

Root Cause

In `programs/liquidity/src/state/token_reserve.rs` (lines 524-530), the supply exchange price update is multiplicative in `Ps`:

```
// line 524-530
supply_exchange_price = supply_exchange_price.safe_add(
    supply_exchange_price
        .safe_mul(supply_rate)?
        .safe_mul(seconds_since_last_update)?
        .safe_div(SECONDS_PER_YEAR.safe_mul(FOUR_DECIMALS)?)?
        .safe_div(FOUR_DECIMALS.safe_mul(FOUR_DECIMALS)?)?,
)?;
```

The `supply_rate` incorporates an amplification factor `ratio_supply_yield` (lines 452-466) that redistributes interest from interest-free positions to interest-bearing ones. At initialization (`Ps = 1e12`), the amplification is calibrated correctly: `S_raw * delta_Ps = B_raw * delta_Pb * (1 - fee)`. But after the first accrual step, `Ps > 1e12`. The formula applies the same amplified rate to the now-larger `Ps`, producing a larger absolute increase on the supply side than what the borrow side generates. Each subsequent step widens the gap.

The key formula at lines 462-466 (when `total_supply_with_interest >= total_supply_interest_free`):

```
// line 462-466
utilization
```

```
        .safe_mul(EXCHANGE_PRICE_RATE_OUTPUT_DECIMALS)?
        .safe_mul(FOUR_DECIMALS.safe_add(supply_ratio)?)?
        .safe_div(FOUR_DECIMALS.safe_mul(FOUR_DECIMALS)?)?
```

Interest conservation requires `Ps(t) = 1e12 + (B_raw / S_raw) * (Pb(t) - 1e12) * (1 - fee)` — a *linear* function of `Pb(t)`, not an exponential. The multiplicative formula forces `Ps` to grow exponentially at a rate higher than `Pb`, violating conservation after the first step.

Attack Scenario

1. A reserve contains both interest-bearing supply (from Lending depositors) and interest-free supply (from Vault positions). This is the protocol's standard operating mode — Vaults always create interest-free Liquidity positions.
2. Borrows remain open and exchange prices are periodically updated (on every user interaction — deposits, withdrawals, borrows, repayments).
3. Because `supply_exchange_price` compounds the amplified rate on top of an already-grown `Ps`, interest-bearing suppliers are credited more value than borrowers pay into the reserve.
4. The deficit accumulates silently with every accrual step, growing faster as `Ps` diverges further from `1e12`.
5. Early withdrawers redeem at inflated exchange prices; remaining suppliers absorb the shortfall when vault balance is insufficient to cover all redemptions.

Impact

- **Reserve deficit**: The total redeemable value promised to interest-bearing suppliers exceeds the assets funded by borrowers. When redemptions occur, the shortfall is borne by later withdrawers.
- **Cumulative and irreversible**: The over-crediting compounds with every exchange-price update, growing larger over time. The protocol has no mechanism to claw back the excess credits.
- **No attacker action required**: This occurs during normal protocol operation whenever interest-free and interest-bearing supply coexist on the same mint — which is the protocol's standard design (Vaults create interest-free positions on the same reserves used by Lending).
- **Accelerates during market stress**: Above the kink (80% utilization), borrow rates jump sharply (from 10% to 150% across the 80-100% range), and the supply-side amplification magnifies the divergence further.

Recommended Mitigation Steps

Compute the supply exchange price increase directly from the actual borrow-side interest collected, distributed only to interest-bearing suppliers:

```
// Correct approach: derive supply increase from borrow increase
let borrow_interest_increase = borrow_exchange_price
    .safe_mul(borrow_rate)?
    .safe_mul(seconds_since_last_update)?
    .safe_div_ceil(SECONDS_PER_YEAR.safe_mul(FOUR_DECIMALS)?)?;

let fee: u128 = self.fee_on_interest.cast()?;
let supplier_share = borrow_interest_increase
    .safe_mul(FOUR_DECIMALS.safe_sub(fee)?)?
    .safe_div(FOUR_DECIMALS)?;

let total_borrow_raw: u128 = self.total_borrow_with_interest.cast()?;
let total_supply_raw: u128 = self.total_supply_with_interest.cast()?;

if total_supply_raw > 0 {
    let price_increase = supplier_share
        .safe_mul(total_borrow_raw)?
        .safe_div(total_supply_raw)?;
    supply_exchange_price =
supply_exchange_price.safe_add(price_increase)?;
}
```

This ensures that the supply-side price increase is always derived from the borrow-side interest actually collected, preserving the conservation invariant regardless of how many accrual steps occur.

Proof of Concept

View detailed Proof of Concept

Sponsor team notes

The Jupiter Lend team considers this finding to be a duplicate of M-10 due to both pertaining to the same type of issue.

[M-03] `get_supply_ratio()` compares raw shares to normal tokens without exchange price conversion, over-allocating yield to with-interest suppliers, leading to insolvency

Submitted by hackhack, also found by abdelhaq16, Auditor3, Bigsam, edoscoba, and pwvux

- `programs/liquidity/src/state/token_reserve.rs` #L96-L109 (`get_supply_ratio` / `get_borrow_ratio`)
- `programs/liquidity/src/state/user_supply_position.rs` #L192-L217 (with-interest stored as raw shares, interest-free stored as normal tokens)

`get_supply_ratio()` is intended to measure the split between with-interest and interest-free supply so the protocol can determine how much borrower interest should be routed to with-interest suppliers. However, the two sides are not stored in the same unit: with-interest supply is tracked in raw shares (`amount * EXCHANGE_PRICES_PRECISION / supply_exchange_price`), while interest-free supply is tracked in normal token units. Every other consumer of `get_total_supply_with_interest()` converts raw shares back to normal tokens before using the value — `get_supply_with_interest_normal_ceil()`, `get_borrow_with_interest_normal()`, and `calc_utilization()` all do this. `get_supply_ratio()` and `get_borrow_ratio()` are the only consumers that skip the conversion.

```
fn get_supply_ratio(&self) -> Result<u128> {
    let supply_ratio: u128 = self.get_with_interest_vs_free_ratio(
        self.get_total_supply_with_interest()?,   // raw shares (not
converted)
        self.get_total_supply_interest_free()?,    // normal tokens
    )?;
    Ok(supply_ratio)
}
```

At deployment (`exchange_price = 1e12`), raw shares and normal tokens are numerically equal, so the comparison is correct. The developer's worked example in the comments (lines 395-409) validates the formula only at this 1:1 price point. As `exchange_price` grows above `1e12`, each raw share represents more tokens than its face value, making the with-interest side appear smaller than it actually is.

This feeds into `calculate_exchange_prices()` where the ratio determines how yield is distributed to with-interest suppliers. When with-interest appears smaller than it is, the `ratio_supply_yield` calculation (lines 452-466) over-concentrates yield: `supply_rate` is set higher than correct, causing `supply_exchange_price` to grow faster than the actual yield generated by borrowers can sustain.

With-interest suppliers can withdraw more tokens than the reserve's yield justifies. Every call to `calculate_exchange_prices()` — triggered by any operate, supply, borrow, or withdraw — compounds the error. The excess comes from the reserve's token balance, creating a

growing insolvency gap. Interest-free depositors or the last with-interest withdrawers absorb the loss. The same bug in `get_borrow_ratio()` has the opposite effect: with-interest borrows appear smaller, so less yield is routed to suppliers than borrowers actually generated. The magnitude scales with exchange price drift — at 36% drift with a 50/50 supply split, the supply_rate is ~15% too high.

Recommended Mitigation Steps

Convert raw shares to normal token amounts before passing to `get_with_interest_vs_free_ratio()`. The conversion functions already exist in the codebase — `get_supply_with_interest_normal_ceil()` and `get_borrow_with_interest_normal()`. Pass the exchange price into the ratio functions:

```
fn get_supply_ratio(&self, supply_exchange_price: u128) -> Result<u128> {
    let with_interest_normal = self.get_total_supply_with_interest()?
        .safe_mul(supply_exchange_price)?
        .safe_div(EXCHANGE_PRICES_PRECISION)?;
    let supply_ratio: u128 = self.get_with_interest_vs_free_ratio(
        with_interest_normal,
        self.get_total_supply_interest_free()?,
    )?;
    Ok(supply_ratio)
}

fn get_borrow_ratio(&self, borrow_exchange_price: u128) -> Result<u128> {
    let with_interest_normal = self.get_total_borrow_with_interest()?
        .safe_mul(borrow_exchange_price)?
        .safe_div(EXCHANGE_PRICES_PRECISION)?;
    let borrow_ratio: u128 = self.get_with_interest_vs_free_ratio(
        with_interest_normal,
        self.get_total_borrow_interest_free()?,
    )?;
    Ok(borrow_ratio)
}
```

The direct branch conditions in `calculate_exchange_prices()` must also be normalized, since they make the same raw-vs-normal comparison: `self.total_supply_with_interest < self.total_supply_interest_free` (line 423) and `self.total_borrow_with_interest < self.total_borrow_interest_free` (line 470). Fixing only the ratio helpers without fixing these branch conditions would leave the pricing logic partially incorrect.

View detailed Proof of Concept

Low Risk and Informational Issues

For this audit, 86 QA reports were submitted by wardens compiling low risk and informational issues. null received the top score from the judge. 46 Low-severity findings were also submitted individually, and can be viewed here.

The following wardens also submitted QA reports: 0x_DyDx, 0x4non, 0xbrett8571, 0xdwrd, 0xFBI, 0xGutzzz, 0xIchigo, 0xIconart, 0xki, 0xMaze, 0xnija, 0xshdax, 7Tecra7, aarontan, AD404, adeolu, AgengDev, Agontuk, amuzetnoM, AuditShield, ayazwx, BadMaths, binbin2803, boodieboodieboo, brianhar, CertiK_security, chuvak, Cold_Exploit, cosin3, Cryptor, darf_tech, Diavolo, Digg3r, Du7sora, edoscoba, eta, EZeek33, HalfBloodPrince, hashmate, I1iveF0rTh1Sh1t, ianyOx, Ibukun, init_state, Internfud, inversive_xyz, InvisibleKT, jaque, jerry0422, johnyfwesh, josephxander, K42, kammiz, Kazopl, kestyvickky, legat, letchupkt, levandr, MAX06200, mikeskynyrd, Miljan9602, moon0x2, mzfr, oct0pwn, opensecretsauce, orangesantra, oziajibogu, peazzycole, pfapostol, qed, reee0001, Rorschach, runup, sarugami, securehash1, shadow0, superpraise, taltcrypto, th3_hybrid, TheCarrot, Tiro, TrinaryMage, vijayn, vt729830, webrainsec, Zbugs, and zcai.

QA Report

Table

ID	DESCRIPTION
[L-01]	Rewards `start_tvl = 0` triggers a division-by-zero and can permanently brick a Lending market when TVL is zero
[L-02]	Guardian `unpause_user` can activate unconfigured positions (NotSet → Active), bricking supply/borrow flows via zeroed config fields
[L-03]	Vault withdrawal-gap check uses floor conversion while Liquidity burns ceil raw units (1-unit limit bypass)
[L-04]	Liquidity claims can be permanently locked by crediting them to a non-signable pubkey (PDA)
[L-05]	Vaults: Liquidation can deterministically revert for specific `debt_amt` values due to strict ref-tick partials monotonicity guard (liveness degradation)
[L-06]	Partial paybacks can hard-revert due to underflow in `Position::get_new_position_info`

ID	DESCRIPTION
[L-07]	Liquidity `max_utilization` can be bypassed due to floor-rounded utilization enforcement
[L-08]	`stop_rewards()` before `start_time` can lock rewards configuration until the scheduled start
[L-09]	Permissionless `lending::update_rate` spam burns rewards accrual via rounding-to-zero
[L-10]	Queued StakePool fee changes can “time-bomb” StakePool-oracle reads (FeeTooHigh after activation)
[L-11]	StakePool oracle fee ceiling can be bypassed due to floor-division rounding
[L-12]	Unprotected “First Initializer Wins” on Core Admin PDAs (Governance Capture on Fresh Deployments)

[L-01] Rewards `start_tvl = 0` triggers a division-by-zero and can permanently brick a Lending market when TVL is zero

Links to root cause

- programs/lendingRewardRateModel/src/state/state.rs:67 (rate gating uses `total_assets < start_tvl`; with `start_tvl = 0`, `total_assets = 0` is not gated)
- programs/lendingRewardRateModel/src/state/state.rs:74 (APR calculation divides by `total_assets`, reverting when `total_assets = 0`)
- `crates/library/src/math/safe_math.rs:79` (`safe_div(0)` returns `LibraryMathError`)
- `programs/lendingRewardRateModel/src/state/context.rs:162` (`start_tvl` is stored with no validation)
- `programs/lending/src/utils/helpers.rs:108` (Lending derives `total_assets` from `f_token_mint.supply`, which is `0` on an empty market)
- `programs/lending/src/utils/helpers.rs:121` (Lending calls `get_rate(total_assets)` unconditionally)
- `programs/lending/src/utils/deposit.rs:74` (deposit calls `update_rates` before minting fTokens, so it cannot “bootstrap” out of `supply = 0`)
- `programs/lendingRewardRateModel/src/state/context.rs:198` (`stop_rewards` CPIs into Lending `update_rate`, which reverts in the bricked state)

Finding description and impact

The Lending program updates its fToken exchange price via `update_rates()`, which incorporates both Liquidity-layer yield and the rewards APR returned by the Lending Reward Rate Model (LRRM). LRRM computes the active APR as `yearly_reward * precision / total_assets`, but only after checking whether `total_assets` is below `start_tvl`.

When `start_tvl` is configured to `0` and the market TVL is `0` (the common case on a fresh market where `f_token_mint.supply == 0`), the gating condition `total_assets < start_tvl` evaluates to false (`0 < 0`), and LRRM executes a division by zero. This deterministically reverts with `LibraryMathError` and aborts the entire instruction.

This is not a cosmetic failure; it bricks core flows:

- The first user deposit cannot succeed because the deposit path calls `update_rates()` before minting any fTokens. Since the transaction reverts before the mint, `f_token_mint.supply` remains `0`, and every subsequent deposit attempt hits the same division-by-zero again.
- The protocol cannot recover on-chain once this configuration is active on an empty market. `stop_rewards` and reward phase transitions CPI into Lending's `update_rate` to sync state; those CPIs also revert in the bricked state, preventing cleanup/reset through the LRRM program.
- Operationally, this can prevent a market from launching (or from reactivating if the market returns to `total_assets == 0` while rewards are configured this way). Remediation requires a program upgrade or an out-of-band migration strategy, both of which are disruptive.

This condition is reachable through privileged configuration (rewards operators/admin auths), but once triggered it becomes a global availability failure for that market.

Recommended mitigation steps

- Fix the arithmetic invariant at the source: treat `total_assets == 0` as a zero-rate case in `LendingRewardsRateModel::get_rate()`, returning `current_rate = 0` (and `next_rate = 0`) without performing any division.
- Add defense-in-depth validation in `start_rewards` to reject `start_tvl == 0` (or to require `start_tvl > 0`) if `0` is not a supported configuration. This prevents accidental footguns even after the arithmetic fix.

[L-02] Guardian `unpause_user` can activate unconfigured positions (NotSet → Active), bricking supply/borrow flows via zeroed config fields

Links to root cause

- programs/liquidity/src/module/admin.rs:823 — unpause_user sets `user_supply_position.status = 0` / `user_borrow_position.status = 0` without

rejecting `configs_not_set()` (NotSet) or requiring a Paused → Active transition.
- programs/liquidity/src/module/admin.rs:779 — `pause_user` explicitly blocks acting when `configs_not_set()` (demonstrates intended state machine).
- `programs/liquidity/src/state/user_supply_position.rs:105` and `programs/liquidity/src/state/user_supply_position.rs:131` — `calc_withdrawal_limit_before_operate()` divides by `expand_duration`; unconfigured positions have `expand_duration = 0`.
- `programs/liquidity/src/state/user_borrow_position.rs:112` and `programs/liquidity/src/state/user_borrow_position.rs:148` — `calc_borrow_limit_before_operate()` divides by `expand_duration`; unconfigured positions have `expand_duration = 0`.

Finding description and impact

`UserSupplyPosition` and `UserBorrowPosition` use `status == NotSet` as the sentinel for "configs have not been initialized yet". When a protocol is onboarded via `init_new_protocol`, the position PDAs are created with `status = NotSet` and the remaining config fields left at their zero defaults (notably `expand_duration = 0`, and the supply side also has `base_withdrawal_limit = 0`).

Only Auths are intended to configure and activate these positions through `update_user_supply_config` / `update_user_borrow_config`, which enforce non-zero config values (including `expand_duration != 0`) and transition the position into an operational state.

However, `unpause_user` is Guardian-callable and can set `status = Active` even when the position is still `NotSet` (unconfigured). This bypasses the "configs must be set before use" gate (and is inconsistent with `pause_user`, which refuses to act when configs are not set). The result is an invalid on-chain state: "Active position with zeroed configuration".

Once a NotSet position is forced Active:

- Supply side: the first deposit can succeed (because `withdrawal_limit == 0` takes an early path), but it sets `withdrawal_limit` non-zero while `expand_duration` remains zero. Any subsequent supply-side interaction that re-enters withdrawal-limit expansion math hits a division-by-zero and reverts (`LIBRARY_MATH_ERROR`), effectively locking funds in the LL position until an Auth repairs the configuration.
- Borrow side: the borrow-limit expansion path divides by `expand_duration` without a comparable early return, so the position can revert immediately on borrow/payback attempts once activated in this broken state.

Operationally, this is a protocol-level DoS/fund-lock footgun reachable by a privileged role that is explicitly *not* responsible for configuring positions. In practice it can prevent child

protocols from completing withdrawals/repayments that depend on LL `operate()` succeeding, until an Auth intervenes to set valid configs.

Recommended mitigation steps

1. Enforce the intended state machine in `unpause_user`:
    - Reject `configs_not_set()` / `status == NotSet`.
    - Require the current status to be Paused before unpausing (i.e., only allow Paused → Active, not NotSet → Active).
2. Add defense-in-depth checks in position logic:
    - In `supply_or_withdraw()` / `borrow_or_payback()`, fail fast if required config fields are invalid (e.g., `expand_duration == 0`), to avoid relying solely on `status` for safety.

[L-03] Vault withdrawal-gap check uses floor conversion while Liquidity burns ceil raw units (1-unit limit bypass)

Links to root cause

- programs/vaults/src/utils/operate.rs:133 (`check_if_withdrawal_safe_for_withdrawal_gap`)
    - programs/vaults/src/utils/operate.rs:152 computes `user_withdrawal` with `safe_div` (floor)
- `programs/liquidity/src/state/user_supply_position.rs:177` (`UserSupplyPosition::supply_or_withdraw`)
    - `programs/liquidity/src/state/user_supply_position.rs:202` converts withdraws with `safe_div_ceil` (ceil)

Finding description and impact

`Vaults::operate()` enforces a protocol-level `withdraw_gap` intended to keep a buffer above Liquidity's dynamic `withdrawal_limit`, so liquidations can still withdraw collateral when users are withdrawing concurrently.

The check converts the user's requested withdrawal amount (normal units) into Liquidity "raw" units and compares it against:

`available_withdrawal - liquidity_withdrawal_limit`.

However, the conversion in `check_if_withdrawal_safe_for_withdrawal_gap()` uses **floor division**:

- `user_withdrawal = floor(withdraw_amount * EXCHANGE_PRICES_PRECISION / liquidity_ex_price)`

Liquidity's actual withdrawal path for `with_interest == 1` supply positions rounds this same conversion **up**:

- `raw_burn = ceil(withdraw_amount * EXCHANGE_PRICES_PRECISION / supply_exchange_price)`

As a result, a withdrawal exists where:

- the vault's gap check passes using the floor-rounded `user_withdrawal`, but
- Liquidity burns `user_withdrawal + 1` raw units (ceil),

meaning the user successfully withdraws **1 raw unit more** than the vault's withdrawal-gap policy allows.

Impact:

- The `withdraw_gap` invariant is violated by a deterministic, user-triggerable off-by-one in raw units.
- In markets where the withdrawal buffer is tight (or where downstream logic assumes strict enforcement), this can erode the intended liquidation buffer and make the protocol more prone to withdrawal-limit friction/DoS during stress.

Recommended mitigation steps

Make Vaults match Liquidity's rounding for `with_interest == 1` withdrawals:

- In `check_if_withdrawal_safe_for_withdrawal_gap()`, compute `user_withdrawal` with `safe_div_ceil` (ceil), not `safe_div` (floor).

This aligns the pre-check with the exact raw amount Liquidity will burn.

[L-04] Liquidity claims can be permanently locked by crediting them to a non-signable pubkey (PDA)

Links to root cause

- programs/liquidity/src/state/context.rs:96 (`init_claim_account` allows initializing a claim account for an arbitrary `user: Pubkey` with no signature from that pubkey)
- programs/liquidity/src/utils/token.rs:8 (`handle_transfer_or_claim()` credits a `UserClaim` based on a caller-provided `claimer: Pubkey` and only checks `claim_account.user == claimer`)
- `programs/liquidity/src/state/context.rs:132` (`claim()` requires `user: Signer` and enforces `claim_account has_one user`, so the claim owner must be able to sign)
- `programs/vaults/src/state/context.rs:481` (`recipient` is unchecked and can be any pubkey, including PDAs)

- `programs/vaults/src/module/user.rs:456` / `programs/vaults/src/module/user.rs:481` (Vaults forwards `recipient` as the Liquidity-layer `withdraw_to` / `borrow_to` when `TransferType::CLAIM` is used)

Finding description and impact

The Liquidity Layer supports `TransferType::CLAIM` as an alternative to direct token transfers: instead of sending tokens immediately from the reserve vault, the protocol credits a `UserClaim` account and increases `TokenReserve.total_claim_amount` to "reserve" those tokens until they are claimed.

However, the claim-owner identity is just a `Pubkey` (`withdraw_to` / `borrow_to`) and is not required to be signable. A claim account can be initialized for any arbitrary `user: Pubkey`, and `operate(..., transfer_type = CLAIM)` can then credit that claim as long as `claim_account.user == claimer`.

Redemption is where the lock occurs: `claim()` requires `user: Signer` and enforces `claim_account has_one user`. If the claim is credited to an off-curve pubkey (e.g., a PDA such as a System Program PDA), that pubkey can never satisfy the `Signer` requirement, so the claim becomes permanently unclaimable.

This has two concrete impacts:

1. **Permanent loss of availability for the credited amount**: the tokens remain held inside the Liquidity vault but remain permanently counted in `total_claim_amount`, making them unusable for future transfers/claims and permanently reducing effective liquidity for that mint.
2. **Direct user loss / stuck withdrawal-borrow proceeds** when an upstream protocol (e.g., Vaults) allows user-controlled recipients in claim mode: a user can end up with funds credited to an address that can never claim them, with no recovery path on-chain.

Recommended mitigation steps

- **Enforce a signable claim owner**: reject off-curve pubkeys as claim owners (e.g., require `withdraw_to` / `borrow_to` to be on-curve when `transfer_type == CLAIM`, and/or require `user` to be on-curve during `init_claim_account`).
- **Harden upstream callers**: for claim-based flows, ensure the claim owner is a wallet-controlled pubkey (or require explicit opt-in checks in Vaults/Lending when a custom recipient is provided with `TransferType::CLAIM`).
- If PDAs must be supported, **change the authorization model** so claim redemption is not gated solely on `user: Signer` for the claim owner (e.g., introduce an explicit claim authority that is required to sign and is guaranteed signable), while preserving the intended security properties.

[L-05] Liquidation can deterministically revert for specific `debt_amt` values due to strict ref-tick partials monotonicity guard (liveness degradation)

Links to root cause

- programs/vaults/src/utils/liquidate.rs:69 — `end_liquidate()` computes `(final_tick, partials)` and enforces the ref-tick partials monotonicity check when `final_tick == ref_tick` on an already-liquidated ref tick.
- programs/vaults/src/state/structs.rs:446 — `CurrentLiquidity::check_is_ref_partials_safe_for_tick()` hard-reverts when `existing_partials > 0 && existing_partials >= partials` (`ErrorCodes::VaultLiquidationReverts`).
- `programs/vaults/src/state/branch.rs:265` — `get_tick_partials()` derives `partials` via integer division from `final_ratio`, making it sensitive to rounding/quantization boundaries.
- `programs/vaults/src/state/structs.rs:515` — `TickMemoryVars::set_partials()` clamps edge cases (`0 → 1`, `>= X30 → X30 - 1`), collapsing distinct ratios into the same partial bucket.

Finding description and impact

The liquidation engine represents progress through price space using a discrete "tick" plus a fractional component ("partials") within the tick. When a liquidation finishes inside a tick, `end_liquidate()` computes `final_ratio`, maps it to a tick interval, derives `partials` with `get_tick_partials()`, and then writes the resulting `(tick, partials)` back to state.

If the liquidation end point lands on a reference tick that is already marked as liquidated (`ref_tick_status == 2`) and `final_tick == ref_tick`, the code treats the ref tick's stored partials as a strict progress marker and enforces that the newly computed partials must be strictly greater than the existing value. The enforcement is implemented as a hard revert:

- `existing_partials > 0 && existing_partials >= partials` ⇒ `VaultLiquidationReverts`

Because `partials` is derived from integer math and then clamped (e.g., `0 → 1`, `>= X30 → X30 - 1`), there are reachable states where multiple nearby `debt_amt` inputs quantize to the same (or a smaller) `partials` bucket. In those cases, liquidation is otherwise feasible (a slightly different `debt_amt` succeeds), but the specific `debt_amt` deterministically reverts during finalization.

Impact:

- **Liquidation liveness degradation**: liquidations can fail for "forbidden" `debt_amt` values even when the vault is clearly liquidatable.

- **Wasted fees / compute:** reverting liquidations consume resources and reduce liquidation throughput under load.
- **Operational risk for fixed-size liquidators:** bots using static chunk sizes can repeatedly hit deterministic reverts and fail to liquidate promptly, increasing the time undercollateralized debt can remain unresolved and raising the probability of delayed liquidations and bad-debt absorption in stressed markets.

Recommended mitigation steps

1. **Remove the hard revert by rounding the end state forward:**
    - If `final_tick == ref_tick` and `partials <= existing_partials`, deterministically "nudge" the end state forward to preserve monotonic progress (e.g., set `partials = min(existing_partials + 1, X30 - 1)`; if already at `X30 - 1`, increment `final_tick` and set `partials = 1`).
2. **Make monotonicity enforcement explicit and user-actionable:**
    - If the guard is retained, perform a pre-check and revert with a dedicated, descriptive error (e.g., "liquidation amount does not advance ref-tick partials; increase `debt_amt`"), rather than reverting deep in finalization with a generic failure.
3. **Align rounding direction to avoid partials regression:**
    - Adjust the rounding in the computations that feed `final_ratio/partials` so the computed end ratio cannot map to a partial bucket that is `<= existing_partials` when finalizing on a liquidated ref tick (i.e., use a conservative monotonic-safe rounding direction in this path).

[L-06] Partial paybacks can hard-revert due to underflow in `Position::get_new_position_info`

Links to root cause

- programs/vaults/src/state/position.rs:178 (partial payback computes `... / borrow_ex_price` then `safe_sub(1)`)

Finding description and impact

`Position::get_new_position_info()` handles partial paybacks (`new_debt < 0 && new_debt != i128::MIN`) by converting the user-supplied payback amount into a raw-debt delta using `floor(abs(new_debt) * EXCHANGE_PRICES_PRECISION / borrow_ex_price)`, then subtracting `1` as part of its rounding convention.

When `borrow_ex_price` is sufficiently large relative to `abs(new_debt) * EXCHANGE_PRICES_PRECISION`, the division evaluates to `0`. The subsequent `safe_sub(1)` underflows and the instruction hard-reverts with a math error. This can occur even when the user's payback amount passes the protocol's minimum operate amount checks.

Impact:

- Certain partial payback amounts become uncallable (deterministic revert), degrading user liveness/UX and preventing incremental risk reduction with those amounts.

Recommended mitigation steps

- Replace the `safe_sub(1)` with a non-underflowing alternative:
  - explicitly handle the `div == 0` case (return a dedicated "amount too small at current exchange price" error), or
  - use `saturating_sub(1)` and reject `payback_amount == 0` with a clear error.
- Align the guardrails with the conversion: enforce that a "valid" partial payback must convert into at least `1` raw unit at the current `borrow_ex_price`, so small values fail gracefully rather than reverting due to arithmetic underflow.

[L-07] Liquidity `max_utilization` can be bypassed due to floor-rounded utilization enforcement

Links to root cause

- programs/liquidity/src/state/token_reserve.rs:586 — `TokenReserve::calc_utilization()`
- programs/liquidity/src/state/token_reserve.rs:629 — utilization uses `safe_div(...)` (floor division)
- programs/liquidity/src/module/user.rs:146 — `operate()` calls `calc_utilization(...)` after updating totals
- `programs/liquidity/src/module/user.rs:154` — borrow reverts only if `utilization > token_reserve.max_utilization`
- PoC test: `tests/src/liquidity/max_utilization_rounding_down_allows_over_borrow_test.rs:16`

Finding description and impact

Liquidity enforces a per-reserve utilization cap (`TokenReserve.max_utilization`, in basis points where `10_000 = 100%`) during `liquidity::operate()` borrows. The check is performed against the value returned by `TokenReserve::calc_utilization()`, which computes:

`utilization = floor(total_scaled_borrow * 10_000 / total_scaled_supply)`.

Because this is a floor-rounded ratio and the borrow guard only reverts when `utilization > max_utilization` (not when the *true* ratio exceeds the cap), a borrower can take a borrow that pushes the *true* utilization slightly above the configured max while the computed (floored) utilization remains `<= max_utilization`.

Concrete example (as exercised in the PoC):

- Configure `max_utilization = 5_000` (50.00%).
- Supply `100_000`.
- Borrow `50_001`.
- True utilization becomes `50_001 / 100_000 = 50.001%`, which is above the configured cap.
- `calc_utilization()` returns `floor(50_001 * 10_000 / 100_000) = floor(5000.1) = 5_000`, so the borrow is accepted.

The bypass is bounded by the utilization precision (1 bp). Even so, it undermines the protocol's risk control by allowing borrows beyond configured limits, reducing the intended liquidity buffer and slightly underestimating utilization for downstream rate calculations at the boundary.

Recommended mitigation steps

Enforce `max_utilization` using a conservative rounding direction for borrows:

- **Ceiling-based check** (simple, local change):
  - Compute `util_ceil = (total_scaled_borrow * 10_000 + total_scaled_supply - 1) / total_scaled_supply`
  - Revert if `borrow_amount > 0 && util_ceil > max_utilization`.
- **Cross-multiplication check** (avoids division/rounding entirely):
  - Require `total_scaled_borrow * 10_000 <= total_scaled_supply * max_utilization`
  - Use `u256`-backed math (or an equivalent helper) to keep the comparison overflow-safe.

Either approach ensures the reserve utilization cap is enforced strictly in the protocol's favor.

Proof of Concept

```
//! PoC: `liquidity::operate()` enforces `TokenReserve.max_utilization`
using a *floored*
//! utilization value.
//!
```

```rust
//! This lets a borrower take an extra sliver of liquidity such that the
*true* utilization is
//! above `max_utilization`, but `calc_utilization()` still returns `<=
max_utilization`.

#[cfg(test)]
mod tests {
    use crate::liquidity::fixture::LiquidityFixture;
    use fluid_test_framework::helpers::MintKey;
    use fluid_test_framework::prelude::*;
    use liquidity::constants::FOUR_DECIMALS;
    use liquidity::state::TokenConfig;

    #[test]
    fn test_max_utilization_rounding_down_allows_over_borrow() {
        let mut fixture = LiquidityFixture::new().expect("Failed to
create fixture");

        let mints = [MintKey::USDC];
        fixture
            .setup_spl_token_mints(&mints)
            .expect("Failed to setup mints");
        fixture.init_liquidity().expect("Failed to init liquidity");
        fixture
            .init_token_reserve(&mints)
            .expect("Failed to init token reserve");
        fixture
            .update_rate_data_v1(&mints)
            .expect("Failed to init rate model");

        // Tighten max utilization to 50% so the boundary is easy to hit
deterministically.
        let max_utilization: u128 = 5_000;
        fixture
            .update_token_config_with_params(
                MintKey::USDC,
                TokenConfig {
                    token: MintKey::USDC.pubkey(),
                    fee: 0,
                    max_utilization,
                },
            )
            .expect("Failed to update token config");

        // Configure a protocol in pure interest-free mode so utilization
is simply `borrow / supply`.
```

```
        let protocol =
fixture.mock_protocol_interest_free.insecure_clone();
        fixture
            .init_new_protocol(&[(MintKey::USDC, MintKey::USDC,
protocol.pubkey())])
            .expect("Failed to init protocol");
        fixture
            .set_user_allowances_default_interest_free(MintKey::USDC,
&protocol)
            .expect("Failed to set protocol allowances");

        let alice = fixture.alice.insecure_clone();
        fixture
            .setup_ata(MintKey::USDC, &alice.pubkey(), 0)
            .expect("Failed to setup Alice ATA");
        fixture
            .setup_ata(MintKey::USDC, &protocol.pubkey(), 0)
            .expect("Failed to setup protocol ATA");
        fixture
            .init_claim_account(MintKey::USDC, &alice.pubkey())
            .expect("Failed to init claim account");

        // Step 1: provide supply.
        //
        // Choose `supply` divisible by `FOUR_DECIMALS` so `max * supply
/ 10_000` is exact.
        let supply: u64 = 100_000;
        fixture
            .mint_to(MintKey::USDC, &alice.pubkey(), supply)
            .expect("Failed to mint USDC to Alice");
        fixture
            .deposit(&protocol, supply, MintKey::USDC, &alice)
            .expect("Failed to deposit");

        // Step 2: borrow 1 unit above the exact 50% threshold.
        //
        // True utilization = 50.001%, but floored utilization = 50.00%,
so the borrow is accepted.
        let borrow: u64 = (supply as u128)
            .checked_mul(max_utilization)
            .unwrap()
            .checked_div(FOUR_DECIMALS)
            .unwrap()
            .checked_add(1)
            .unwrap() as u64;
        fixture
```

```
            .borrow(&protocol, borrow, MintKey::USDC, &alice)
            .expect("Borrow unexpectedly reverted");

        // Sanity: the reserve reports utilization exactly at the
configured max.
        let reserve = fixture
            .read_token_reserve(MintKey::USDC)
            .expect("Failed to read reserve");
        assert_eq!(
            reserve.last_utilization as u128, max_utilization,
            "Precondition: utilization is floored to the max"
        );

        // Step 3: demonstrate the breach: `borrow / supply` is strictly
above `max_utilization`.
        let total_supply: u128 = reserve.total_supply_interest_free as
u128;
        let total_borrow: u128 = reserve.total_borrow_interest_free as
u128;
        assert_eq!(total_supply, supply as u128, "Expected interest-free
supply only");
        assert_eq!(total_borrow, borrow as u128, "Expected interest-free
borrow only");
        assert!(
            total_borrow
                .checked_mul(FOUR_DECIMALS)
                .unwrap()
                > max_utilization.checked_mul(total_supply).unwrap(),
            "Expected true utilization to exceed the configured max"
        );

        // If utilization were computed conservatively (ceiling), it
would be 50.01% and the borrow should revert.
        let util_ceil = total_borrow
            .checked_mul(FOUR_DECIMALS)
            .unwrap()
            .checked_add(total_supply - 1)
            .unwrap()
            .checked_div(total_supply)
            .unwrap();
        assert_eq!(
            util_ceil,
            max_utilization + 1,
            "Ceiling utilization should exceed max by 1 bp"
        );
```

```
        // Step 4: borrowing any additional amount now reverts as
expected once the *floored* value crosses the limit.
        fixture.expect_revert_with("MaxUtilizationReached", |f| {
            f.borrow(&protocol, 10, MintKey::USDC, &alice)
        });
    }
}
```

[L-08] `stop_rewards()` before `start_time` can lock rewards configuration until the scheduled start

Links to root cause

- programs/lendingRewardRateModel/src/state/context.rs:174 (`LendingRewards::stop`)
- programs/lendingRewardRateModel/src/state/context.rs:101 (`LendingRewards::start`)

Finding description and impact

`stop_rewards()` is callable while rewards are only *scheduled* (i.e., `current_time < start_time`). In that case, `stop()` updates the model using saturating arithmetic:

- `duration = current_time.saturating_sub(start_time).saturating_sub(1);`

When `current_time < start_time`, this sets `duration` to `0` while leaving `start_time` unchanged (still in the future). From that point, `start_rewards()` computes `end_time = start_time + duration = start_time` and rejects any new configuration as long as `current_time <= end_time` (reverts `NotEnded`). Practically, the rewards configuration becomes non-overwritable until the original future `start_time` passes.

This creates an operational DoS on rewards configuration:

- A whitelisted rewards configurator can accidentally "cancel" a future rewards schedule and unintentionally lock the market's rewards parameters for days/weeks/months (depending on how far in the future `start_time` was set).
- A malicious or compromised configurator key can intentionally grief incentives by calling `stop_rewards()` before `start_time`, preventing immediate correction.

The issue does not directly expose funds, but it can materially disrupt incentive operations and any downstream processes that depend on timely rewards reconfiguration.

Recommended mitigation steps

- Treat “stop before start” as a true cancellation. If `current_time < start_time`, reset the active schedule instead of setting `duration` to `0`:
  - clear `start_time`, `duration`, and `yearly_reward` (and consider clearing any queued-next fields if present), or
  - add a dedicated `cancel_scheduled_rewards()` instruction and make `stop_rewards()` require `current_time >= start_time`.
- Avoid saturating math for `duration` updates in `stop()`. Explicitly branch on `current_time < start_time` / `current_time >= start_time` to ensure the post-state is consistent and can be reconfigured immediately.

[L-09] Permissionless `lending::update_rate` spam burns rewards accrual via rounding-to-zero

Links to root cause

- programs/lending/src/state/context.rs:513 (the `UpdateRate` accounts struct has no signer; instruction is permissionless)
- programs/lending/src/module/admin.rs:224 (`update_rate()` entrypoint exposes `UpdateRate` publicly)
- programs/lending/src/utils/helpers.rs:94 (exchange-price update uses `lending.last_update_timestamp` as the accrual start)
- `programs/lending/src/utils/helpers.rs:148` (rewards return uses integer division by `SECONDS_PER_YEAR`, truncating small time slices to zero)
- `programs/lending/src/utils/helpers.rs:208` (token exchange-price delta uses integer division by `RETURN_PERCENT_PRECISION`, so small per-update returns round down to 0)
- `programs/lending/src/utils/helpers.rs:219` + `programs/lending/src/utils/helpers.rs:232` + `programs/lending/src/utils/helpers.rs:234` (`update_rates()` writes `token_exchange_price` and unconditionally advances `last_update_timestamp`, even if the computed exchange price did not increase)
- `programs/lending/src/constant.rs:4` + `programs/lending/src/constant.rs:6` (precision constants that make the rounding regime reachable)

Finding description and impact

The `lending::update_rate` instruction is permissionless and can be called by any external account. Each call computes the new `lending.token_exchange_price` from (i) the Liquidity-layer supply exchange price delta and (ii) rewards-return computed from the rewards rate model over `time_diff = now - lending.last_update_timestamp`.

Both components rely on fixed-point integer math with floor division. When the per-call return is small enough, the computed exchange-price increment truncates to zero (i.e., `new_token_exchange_price == old_token_exchange_price`). Despite that, `update_rates()` still persists `lending.last_update_timestamp = Clock::get()?.unix_timestamp`.

This combination creates a griefing vector: an attacker can call `update_rate` at a high cadence (e.g., every second) to keep `time_diff` small enough that the rewards-return (and therefore the exchange-price increment) repeatedly rounds to zero. Because `last_update_timestamp` is advanced on every call, the "fractional" return from each small interval is discarded rather than carried forward, permanently reducing the rewards credited to all fToken holders. The effect persists as long as the attacker continues to spam updates, and it does not require any special privileges—only transaction fees.

Recommended mitigation steps

Carry fractional accrual forward instead of discarding it when an update rounds down to zero. Concretely:

- Add a remainder accumulator in `Lending` (e.g., `pending_return_scaled` / `exchange_price_remainder`) and compute `delta = (old_price * total_return + remainder) / RETURN_PERCENT_PRECISION`, updating `remainder = (old_price * total_return + remainder) % RETURN_PERCENT_PRECISION`.
- As a minimal fix if no accumulator is added: do not advance `last_update_timestamp` (and avoid committing the new `liquidity_exchange_price`) when the computed `token_exchange_price` delta is zero, so elapsed time cannot be "burned" by high-frequency calls.

[L-10] Queued StakePool fee changes can "time-bomb" StakePool-oracle reads (FeeTooHigh after activation)

Links to root cause

- programs/oracle/src/modules/stake_pool.rs:50 — `check_fee()` hard-reverts with `FeeTooHigh` when a *current* StakePool fee exceeds the oracle ceiling.
- programs/oracle/src/modules/stake_pool.rs:65 — `check_future_fee()` detects an above-ceiling *queued* fee but only emits `LogStakePoolHighFeeDetected` and returns `Ok(())`.
- programs/oracle/src/modules/stake_pool.rs:91 — `is_fee_too_high()` enforces current fees but does not block queued above-ceiling fees.
- `programs/oracle/src/constants.rs:21` — `MAX_FEE_CEILING` defines the 0.5% (per-mille) fee ceiling.
- `programs/vaults/src/invokes/oracle.rs:37` — any Oracle CPI failure is surfaced to Vaults as `VaultCpiToOracleFailed`, halting dependent flows.

Finding description and impact

The StakePool oracle hop (`read_stake_pool_source`) is designed to reject SPL stake-pools whose fee configuration is incompatible with the protocol's pricing assumptions. It does this by scaling each *current* deposit/withdrawal fee and reverting with `FeeTooHigh` when any scaled fee exceeds `MAX_FEE_CEILING` (0.5%).

However, SPL stake-pool fee updates are staged through `FutureEpoch` (queued into `next_*_withdrawal_fee` and later promoted into the corresponding current fee during an epoch refresh). The oracle explicitly parses and inspects these queued fee values, but it treats them as advisory: `check_future_fee()` emits `LogStakePoolHighFeeDetected` and still returns a price.

This creates a predictable failure mode for any market using a StakePool hop:

- A StakePool manager can queue a withdrawal fee above the oracle ceiling while oracle reads remain successful.
- When that queued fee becomes current (after StakePool refresh at a later epoch boundary; SPL stake-pool uses a two-epoch delay), the oracle begins hard-reverting with `FeeTooHigh`.
- Downstream programs that require oracle reads (e.g., Vault `operate()` / `liquidate()`) will revert because the Oracle CPI fails, freezing market functionality for any position that depends on that oracle hop until the oracle configuration is updated to remove or replace the StakePool source.

The "time-bomb" aspect is that the market can appear healthy (oracle returning a price) even after the on-chain data already contains a queued state transition that will deterministically brick oracle reads once applied.

Recommended mitigation steps

Make queued above-ceiling fees an on-chain invariant violation, not a warning:

- Change `check_future_fee()` to return an error (optionally emitting `LogStakePoolHighFeeDetected` immediately before reverting) when a `FutureEpoch::One` or `FutureEpoch::Two` fee scales above `MAX_FEE_CEILING`.
- This removes the "works now, bricks later" behavior by forcing the system to treat a scheduled incompatible fee change as immediately unsupported, requiring the oracle configuration to be updated as soon as the incompatible change is observed.

[L-11] StakePool oracle fee ceiling can be bypassed due to floor-division rounding

Links to root cause

- programs/oracle/src/modules/stake_pool.rs:50 — `check_fee()` scales fees as `numerator * 1000 / denominator` using floor division.
- programs/oracle/src/modules/stake_pool.rs:58 — ceiling enforcement only checks `fee_scaled > MAX_FEE_CEILING`.
- `programs/oracle/src/constants.rs:20` — `MAX_FEE_CEILING` is documented as a 0.5% maximum (5 per-mille).

Finding description and impact

The StakePool oracle enforces a maximum acceptable fee using an integer per-mille scale. It computes:

```
fee_scaled = fee.numerator * 1000 / fee.denominator
```

and reverts only when `fee_scaled > MAX_FEE_CEILING` (5). Because the division is floored, any fee in the range:

```
[0.500%, 0.599%...]
```

will still yield `fee_scaled == 5` and therefore be accepted as if it were at or below 0.5%.

For example, a fee of `599 / 100000 = 0.599%` produces: `599 * 1000 / 100000 = 5` (floor)

so the oracle treats it as compliant even though it exceeds the advertised maximum.

Impact:

- The oracle's fee ceiling is not enforced precisely; stake pools with materially higher withdrawal/deposit fees than intended can remain supported.
- Since the StakePool price computation derives an exchange rate from `total_lamports / pool_token_supply` without applying withdrawal/deposit fees, accepting higher-fee stake pools widens the discrepancy between the oracle's implied redemption value and the real value users receive after fees.
- This increases pricing error in any market using a StakePool hop, weakening the protocol's assumptions around "supported" stake pools and potentially worsening risk parameters that rely on the fee ceiling as a guardrail.

Recommended mitigation steps

Enforce the ceiling without relying on floor-division rounding:

- Replace the scaled computation/compare with a cross-multiplication check (or a ceil-division check) that preserves the intended bound, e.g.:

```
fee.numerator * 1000 <= fee.denominator * MAX_FEE_CEILING
```

This ensures any fee strictly above 0.5% is rejected, including values that previously slipped through due to truncation.

[L-12] Unprotected “First Initializer Wins” on Core Admin PDAs (Governance Capture on Fresh Deployments)

Links to root cause

- programs/liquidity/src/state/seeds.rs:5 (constant `LIQUIDITY_SEED`)
- programs/liquidity/src/state/seeds.rs:6 (constant `AUTH_LIST_SEED`)
- programs/liquidity/src/state/context.rs:169 (`InitLiquidity` is permissionless; no authority allowlist)
- `programs/liquidity/src/module/admin.rs:22` (`init_liquidity` stores caller-chosen `authority` and initializes `auth_list`)
- `programs/liquidity/src/state/state.rs:21` (`Liquidity::init` writes `authority`)
- `programs/liquidity/src/state/state.rs:49` (`AuthorizationList::init` auto-whitelists `authority`)
- `programs/liquidity/src/state/context.rs:275` (`UpdateAuthority` requires current `liquidity.authority` signer)
- `programs/liquidity/src/module/admin.rs:119` (`update_authority` enforces signer == current authority; no recovery without that key)
- `programs/lending/src/state/seeds.rs:3` (constant `LENDING_ADMIN_SEED`)
- `programs/lending/src/state/context.rs:22` (`InitLendingAdmin` is permissionless; no authority allowlist)
- `programs/lending/src/module/admin.rs:17` (`init_lending_admin` stores caller-chosen `authority` and whitelists it)
- `programs/lending/src/state/context.rs:151` (`UpdateAuthority` requires current `lending_admin.authority` signer)
- `programs/lending/src/module/admin.rs:183` (`update_authority` blocks recovery without current authority signature)

Finding description and impact

Both the Liquidity and Lending programs rely on singleton “admin” PDAs whose addresses are derived from constant, public seeds:

- Liquidity: `liquidity` PDA (`LIQUIDITY_SEED`) and `auth_list` PDA (`AUTH_LIST_SEED`)
- Lending: `lending_admin` PDA (`LENDING_ADMIN_SEED`)

The initialization instructions for these singleton PDAs are permissionless. Any signer can create them first and set the stored authority to an arbitrary pubkey supplied as an instruction argument:

- Liquidity: `init_liquidity(authority, revenue_collector)` writes `liquidity.authority = authority` and initializes `auth_list` such that `authority` becomes the first whitelisted auth/guardian.
- Lending: `init_lending_admin(..., authority)` writes `lending_admin.authority = authority` and pushes `authority` into `lending_admin.auths`.

This creates a straightforward deployment-time takeover:

1. On a fresh deployment (new program id / new cluster), the admin PDAs do not exist.
2. A third party calls the permissionless init instruction first and sets `authority` to themselves.
3. From that point on, governance cannot recover on-chain because all authority rotation paths require the *current* authority signature.

Impact:

- Permanent governance capture of the Liquidity admin surface: the attacker becomes the sole authority and first auth/guardian, and can subsequently grant/revoke auths, change protocol status (lock/unlock), list and configure reserves, set rate curves and token configs, configure protocol borrow/supply limits, and control revenue collection flows.
- Permanent governance capture of the Lending admin surface: the attacker becomes the authority and an auth, controlling auth membership and operational parameters gated behind those checks (including market initialization flows gated by `auths`).
- In practice this blocks a safe launch on any fresh deployment and, if the protocol is launched in a compromised state, it gives the attacker administrative control over the configuration that governs user funds.

The sponsor documentation describes intended “Program Authority” and “Init Authority” keys (`README-sponsor.md`), but these keys are not enforced by the on-chain init constraints for the singleton admin PDAs.

Recommended mitigation steps

Make singleton admin PDA initialization non-capturable by constraining the initializer and removing attacker-controlled authority assignment:

1. **Restrict initializer signer**:
    - Add an explicit signer constraint (e.g., `#[account(address = GOVERNANCE_MS)]`) to `InitLiquidity.signer` and `InitLendingAdmin.authority`, or alternatively gate to `PROTOCOL_INIT_AUTH` if you intentionally separate init vs governance.
2. **Eliminate the “set authority by argument” pattern**:
    - Do not accept `authority: Pubkey` as an init argument for these singleton PDAs.

- Write `authority` from the constrained signer (or hardcode to `GOVERNANCE_MS`) so a non-governance signer cannot install themselves even if they manage to call the instruction.

Disclosures

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.