

Security Assessment Report



Fluid Liquidity Layer

August 2026

Prepared for Fluid / Mysten Labs

Table of Contents

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

Assessment Methodology **6**

Threat and Security Overview **7**
- Assets 7
- Actors 7
- Trust Assumptions 8
- Attack Vectors 8
- Findings Summary 9
- Severity Matrix 9

Detailed Findings **10**
- Low Severity Issues 12
  - L-01: Post-accrual BEP usage in calculate_exchange_prices can cause supplier overcredit and reserve insolvency 12
  - L-02: Repeated zero-net supply cycles extend withdrawal-limit decay 15
  - L-03: Base withdrawal-limit updates do not reconcile the active withdrawal limit 18
  - L-04: Zero-supply borrowing turns collectable protocol revenue into zero-utilisation-rate credit exposure. 21
- Informational Severity Issues 23
  - I-01: Missing event emission for TokenReserve initialization 23
  - I-02: Revenue collection leaves the reserve in an inconsistent update state 24
  - I-03: Subtracting the floored interest-free share rounds supplier yield up 26
  - I-04: Utilization checks reuse the max_rate constant 28
  - I-05: Expansion parameters are applied retrospectively after config updates 29
  - I-06: Withdrawals can push utilization above max_utilization 31
  - I-07: Borrow mode-switch conversion can leave amount > debt_ceiling, violating the borrow-limit invariant 32
  - I-08: Dust ratio early return accrues borrow EP while freezing supply EP, diverting essentially all WI borrower interest to revenue 33
  - I-09: UserCaps can be minted for unlisted tokens 34
  - I-10: Rounded mixed-mode supply ratios can over-credit WI suppliers 35
  - I-11: Zero object IDs can be registered as valid holders 37
  - I-12: Single-admin key is total governance — no quorum or threshold anywhere 39
  - I-13: Global pause is auth-controlled in both directions, so one compromised auth can freeze or unilaterally unfreeze all operate legs 40
  - I-14: Sub-minimum residual positions cannot be settled through ordinary single-leg operations 41

Disclaimer **43**

About Certora **43**

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Fluid Liquidity Layer	https://github.com/Instadapp/fluid-packages-sui	Initial: c9b3bd3 Final: a79be0e	Sui / Move

Project Overview

This document describes the security review of **Fluid Liquidity Layer**. The work was undertaken from **August 3rd, 2026** to **August 19th, 2026**.

The following contract list is included in our scope:

packages/library/sources/**/*.move
packages/liquidity/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

Fluid is a modular DeFi protocol developed by Instadapp around a shared liquidity foundation. Instead of isolating capital across separate products, its architecture allows multiple protocols to draw from the same reserves while maintaining their own accounting and risk controls. On Ethereum, Fluid supports lending markets, collateralized vaults, and decentralized exchange functionality. On Solana, the same model has been implemented through Jupiter Lend, with dedicated programs for liquidity, lending, vaults, oracles, and flash loans.

At the center of the Sui implementation is the Liquidity Layer, which provides the system's shared accounting and custody infrastructure. It holds token reserves, records authorized protocols' supply and borrow positions, accrues interest through exchange prices, and enforces

utilization, withdrawal, and borrowing limits. End users do not interact with it directly; registered protocols route supply, withdrawal, borrowing, and repayment operations through a common interface. This structure reduces liquidity fragmentation and allows capital to be shared across the wider Fluid system. Lending, vault, flash-loan, and other protocol modules are intended to be built on top of this foundation.

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 Liquidity Layer is the shared custody and accounting foundation through which registered Fluid protocols supply, withdraw, borrow, and repay listed coins. Each `TokenReserve<T>` holds pooled assets and tracks protocol positions through exchange prices, utilization, withdrawal limits, decay, and debt ceilings. End users interact indirectly through registered protocols holding `UserCap<T>` objects, while privileged roles configure and protect the system. Its main security boundaries are reserve solvency, capability custody, privileged configuration, arithmetic consistency, and token availability. The following threat model summarizes the assets, actors, assumptions, and non-exhaustive attack vectors considered during the review.

Assets

- **Reserve vault balances** - pooled coins held for protocol operations.
- **Supplier positions** - principal and accrued yield claims.
- **Borrower positions** - principal and accrued debt obligations.
- **Protocol revenue** - fees and collectable reserve surplus.
- **Available liquidity** - funds supporting withdrawals and new borrows.
- **Exchange-price state** - shared supply and debt valuation indices.
- **Risk-limit state** - withdrawal floors, decay, and debt ceilings.
- **UserCap authority** - control over protocol positions and receivers.
- **Protocol configuration** - reserve listings, roles, pauses, and upgrades.

Actors

- **Protocol governance** - manages roles, upgrades, authorization, and UserCaps.
- **Auth operators** - configure reserves, rates, limits, and revenue collection.
- **Guardians** - pause token and protocol operations during incidents.
- **Registered protocols** - hold UserCaps and operate shared positions.
- **Upstream users** - supply or borrow through registered protocols.
- **Permissionless keepers** - refresh exchange prices, utilization, and rates.
- **Revenue collector** - receives fees withdrawn from reserve vaults.

- **Coin issuers**

Trust Assumptions

- **UserCap custody** - registered protocols prevent loss or unauthorized use.
- **Integration correctness** - registered protocols enforce upstream collateral and accounting.
- **Token availability** - listed coins remain transferable and issuer-supported.
- **Sui correctness** - object, coin, clock, and transaction semantics hold.
- **Safe configuration** - rates, fees, limits, and ceilings remain conservative.
- **Supported modes** - reserves mainly operate under with-interest mode.
- **Update liveness** - reserve state receives timely permissionless or operational updates.
- **Upgrade integrity** - package upgrades preserve state and reviewed security invariants.

Attack Vectors

- **Multi-leg operation coupling** - one leg masks restrictions affecting another.
- **Regulated-coin controls** - issuers pause or deny-list reserve operations.
- **UserCap lifecycle abuse** - theft, sharing, freezing, loss, or invalid registration.
- **Mixed-mode accrual mismatch** - overcredits suppliers beyond reserve assets.
- **Minimum-operation dust** - blocks final withdrawals and repayments.
- **Zero-supply borrowing** - exposes retained revenue at minimum rates.
- **Withdrawal-decay cycling** - extends limits after supply-neutral round trips.
- **Base-limit reconfiguration** - preserves stale state across boundary changes.
- **Expansion reconfiguration** - applies new parameters to elapsed historical time.
- **Utilization transitions** - exceed limits during withdrawals or recapitalization.
- **Precision loss** - biases yield, mode conversions, and debt ceilings.
- **Privileged-role compromise** - halts exits, reopens incidents, or redirects revenue.
- **State-observability gaps** - produce stale snapshots and missed reserve initialization.
- **Upstream collateral failure** - protocols create debt without adequate external backing.
- **Shared-liquidity contagion** - one protocol's bad debt harms other suppliers.
- **Receiver redirection** - compromised integrations route withdrawals or borrows elsewhere.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	4	4	1
Informational	14	14	7
Total	18	18	8

Severity Matrix

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

Detailed Findings

ID	Title	Severity	Status
L-01	Post-accrual BEP usage in calculate_exchange_prices can cause supplier overcredit and reserve insolvency	Low	Fixed
L-02	Repeated zero-net supply cycles extend withdrawal-limit decay	Low	Acknowledged
L-03	Base withdrawal-limit updates do not reconcile the active withdrawal limit	Low	Acknowledged
L-04	Zero-supply borrowing turns collectable protocol revenue into zero-utilisation-rate credit exposure	Low	Acknowledged
I-01	Missing event emission for TokenReserve initialization	Informational	Fixed
I-02	Revenue collection leaves the reserve in an inconsistent update state	Informational	Acknowledged
I-03	Subtracting the floored interest-free share rounds supplier yield up	Informational	Fixed
I-04	Utilization checks reuse the max_rate constant	Informational	Acknowledged

I-05	Expansion parameters are applied retrospectively after config updates	Informational	Fixed
I-06	Withdrawals can push utilization above max_utilization	Informational	Acknowledged
I-07	Borrow mode-switch conversion can leave amount > debt_ceiling, violating the borrow-limit invariant	Informational	Acknowledged
I-08	Dust ratio early return accrues borrow EP while freezing supply EP, diverting essentially all WI borrower interest to revenue	Informational	Fixed
I-09	UserCaps can be minted for unlisted tokens	Informational	Fixed
I-10	Rounded mixed-mode supply ratios can over-credit WI suppliers	Informational	Fixed
I-11	Zero object IDs can be registered as valid holders	Informational	Fixed
I-12	Single-admin key is total governance — no quorum or threshold anywhere	Informational	Acknowledged
I-13	Global pause is auth-controlled in both directions, so one compromised auth can freeze or unilaterally unfreeze all operate legs	Informational	Acknowledged
I-14	Sub-minimum residual positions cannot be settled through ordinary single-leg operations	Informational	Acknowledged

Low Severity Issues

L-01: Post-accrual BEP usage in calculate_exchange_prices can cause supplier overcredit and reserve insolvency

Severity: Low	Impact: High	Likelihood: Low
Files: packages/liquidity/sources/state/reserve.move	Status: Fixed	

Description:

`calculate_exchange_prices` updates the borrow exchange price before calculating the proportion of total borrowing that pays interest:

```
borrow_exchange_price = borrow_exchange_price + accrued_interest;

let borrow_with_interest_normal =
    total_borrow_with_interest
        * borrow_exchange_price
        / exchange_price_precision;
```

The updated borrow exchange price represents the with-interest debt at the end of the accrual period. However, the resulting borrow ratio is used to calculate supplier yield for the entire period.

Consequently, interest accumulated during the period is treated as though it existed at its beginning. When both with-interest and interest-free borrowings exist, this overstates the proportion of borrowers paying interest and causes the supply exchange price to accrue more interest than borrowers generated.

For example, consider the following market:

```
With-interest supply:  10,000
With-interest borrow:   8,000
Interest-free borrow:   2,000
```

```
Borrow rate:              150%
Protocol fee:               5%
Elapsed time:             1 year
```

After one year, the with-interest debt is:

```
8,000 × 2.5 = 20,000
```

The reserve therefore has `22,000` in total accounting assets:

```
20,000 with-interest debt
+2,000 interest-free debt
= 22,000 total assets
```

However, the implementation calculates the paying-borrower proportion using the updated debt:

```
20,000 / (20,000 + 2,000) ≈ 90.9%
```

This increases the supply exchange price from `1e12` to `2.295325e12`, resulting in:

```
Supplier claims:     22,953.25
Reserve assets:      22,000.00
Accounting deficit:     953.25
```

The with-interest borrowers generated only `12,000` of gross interest, but suppliers were credited with `12,953.25`. Supplier claims therefore exceed the reserve's assets, making the reserve insolvent.

The revenue calculation cannot correct this deficit because negative revenue is treated as zero. Even if every borrower fully repays, the reserve receives only `22,000`, while suppliers hold claims for `22,953.25`. The permanent shortfall will ultimately result in failed or socialised withdrawals.

Using the old stored borrow exchange price would instead calculate an `80%` paying-borrower proportion. The `12,000` of borrower interest would then be distributed correctly:

```
Protocol revenue:   12,000 * 5%  =    600
Supplier interest:  12,000 * 95% = 11,400
Supplier claims:    10,000 + 11,400 = 21,400
```

Note: The issue requires persistent mixed-mode borrowing with a nonzero interest-free balance. Current reserves are launched with-interest-only, so the affected calculation is not exercised

under the intended configuration. However, the code does not inherently prevent this configuration from being enabled later.

Recommendations:

Consider using the pre-accrual borrow exchange price when normalizing the with-interest borrow amount so that the borrower proportion, utilization, and borrow rate refer to the same reserve state. Alternatively, if interest-free borrowing will remain unsupported, consider enforcing the with-interest-only configuration on-chain or documenting mixed-mode borrowing as unsupported and ensuring it cannot be enabled operationally.

Customer Response:

Fixed in commit 7e470fb.

Fix Review:

Fixed.

L-02: Repeated zero-net supply cycles extend withdrawal-limit decay

Severity: Low	Impact: Medium	Likelihood: Low
Files: packages/liquidity/sources/state/supply_position.move	Status: Acknowledged	

Description:

When a deposit creates new decay while decay already exists, the protocol combines their remaining durations using an amount-weighted average:

```
merged duration =
(existing duration * existing decay + 3600 * new decay)
/ (existing decay + new decay)
```

This moves the existing decay toward a fresh 3,600-second duration. If the deposited amount is immediately withdrawn, the withdrawal logic reverses the corresponding decay amount but keeps the newly blended duration.

An authorized caller can therefore repeat an equal deposit-and-withdraw cycle. Each cycle restores the supply amount, withdrawal limit, and decay amount to their previous values while extending the time over which the existing decay remains active.

For example, consider the following position using raw token units:

```
Base withdrawal limit: 100
Expansion percentage:  20%
Initial supply:         600
```

The fully expanded withdrawal limit is 480, so the initial deposit creates:

```
Withdrawal limit: 480
Decay amount:     480 - 100 = 380
Decay duration:   3600 seconds
```

After 1,800 seconds, half of the decay has been consumed:

```
Decay amount:   190
Decay duration: 1800 seconds
```

The caller now deposits `150`. Supply becomes `750`, whose fully expanded withdrawal limit is `600`, creating `120` of new decay. The duration becomes:

```
(1800 * 190 + 3600 * 120) / (190 + 120) = 2496 seconds
```

Immediately withdrawing the same `150` restores the original amount-related state:

```
Supply:           600
Withdrawal limit: 480
Decay amount:     190
Decay duration:   2496 seconds
```

Although the deposit has been fully recovered, the decay duration has increased from `1800` to `2496` seconds. Repeating the cycle at the same timestamp progressively extends it:

```
Completed cycles:  1
Remaining duration: 2496 seconds

Completed cycles:  2
Remaining duration: 2923 seconds

Completed cycles: 10
Remaining duration: 3585 seconds
```

At the original 3,600-second expiry from the first deposit, ten cycles leave `95` decay:

```
consumed decay  = floor(190 * 1800 / 3585) = 95
remaining decay = 190 - 95 = 95
```

The position can then withdraw `120`, which consumes the remaining decay and lowers the stored withdrawal limit, followed by another withdrawal of `95`. Its aggregate withdrawal capacity at that timestamp is therefore `215`, compared with `120` under the original decay schedule.

This happens because deposits update both the decay amount and its duration, while withdrawals reverse only the amount. A flash deposit can leave the position's amounts unchanged but still extend how long its existing decay remains active. This preserves additional withdrawal capacity beyond the original schedule and allows reserve liquidity to be removed

earlier. The same behavior exists in the EVM and Solana implementations, although precision differences may slightly change the rounded values.

Recommendations:

A complete fix is non-trivial because merging decay amounts into one duration loses the information needed to reverse an individual deposit. Consider one of the following approaches:

1. Track decay in bounded per-deposit tranches and consume the newest tranche first during withdrawals. This preserves each deposit's schedule and makes deposit-withdraw round trips reversible, but increases storage and computation and requires a policy for limiting or merging tranches.

2. When a withdrawal reduces the decay amount, reduce its remaining duration proportionally:

```
new duration = old duration × new decay / old decay
```

This requires no additional persistent state and prevents a smaller decay amount from retaining an extended duration. However, it is conservative: legitimate decay may expire earlier than intended, repeated withdrawals may shorten it further, and rounding to zero must be handled.

3. If the behavior is accepted, document that supply-neutral deposit-withdraw cycles can prolong active decay and permit earlier withdrawals than the original schedule. This should be treated as an explicit trust assumption for authorized callers rather than an implementation fix.

Customer Response:

Acknowledged - *"The decay clamp caps how much capacity cycling can ever pull forward (max 36.8%, decaying fast after), it can't touch other users' funds or protocol solvency, the same behavior already exists in the live EVM/Solana contracts."*

L-03: Base withdrawal-limit updates do not reconcile the active withdrawal limit

Severity: Low	Impact: Medium	Likelihood: Low
Files: packages/liquidity/sources/module/admin.move packages/liquidity/sources/state/supply_position.move packages/liquidity/sources/math/liquidity_calcs.move	Status: Acknowledged	

Description:

A supply position may withdraw its entire balance when its supply amount is below its configured `base_withdrawal_limit`. This state is represented by a stored withdrawal limit of zero:

```
if (last_limit == 0) { return 0 };
```

When `update_user_supply_config` changes the base withdrawal limit without changing the position's mode, it preserves the stored withdrawal limit. The subsequent pre-operation calculation does not receive the new base and therefore cannot determine whether the stored limit remains valid.

Consider:

```
Supply amount:                150
Old base withdrawal limit:    200
Stored withdrawal limit:      0
Expand percentage:            20%
```

The admin then lowers the base to `100` without changing the expansion percentage or duration. Under the new configuration, the position is above its base and should retain:

```
150 - floor(150 * 20%) = 120
```

Nevertheless, the next withdrawal sees the preserved zero and returns from `calc_withdrawal_limit_before_operate` immediately. The holder can withdraw its complete `150` supply position instead of being limited to `30`.

The reverse transition also leaves a stale state. If a position has supply `150`, a stored limit of `120`, and its base is raised from `100` to `200`, the position should become fully withdrawable. Instead, the pre-operation calculation continues enforcing the stored limit until a successful operation updates it or the administrator explicitly updates the withdrawal limit.

Note: This behavior is independent of the retrospective application of expansion parameters. It occurs with unchanged expansion parameters, requires no elapsed time, and is not fixed by refreshing `last_update`. The borrow-side calculation receives its base limit directly, so the same behavior does not apply to borrow positions. This behavior is inherited from the EVM and Solana implementations.

Recommendations:

Consider using one of the following approaches:

1. Make `update_user_supply_config` update the stored withdrawal limit when the position crosses the base boundary:

```
was_below_old_base = old_amount < old_base
is_below_new_base  = new_amount < new_base

if was_below_old_base and not is_below_new_base:
    minimum_limit =
        new_amount
        - floor(new_amount * new_expand_percent / 10,000)

    stored_limit = max(stored_limit, minimum_limit)

else if not was_below_old_base and is_below_new_base:
    stored_limit = 0

else:
    preserve stored_limit
```

2. Call `update_user_withdrawal_limit` with `0` immediately after `update_user_supply_config` in the same PTB whenever the position crosses the base boundary.

Customer Response:

Acknowledged - *"The exposure window is exactly one operation wide: calc_withdrawal_limit_after_operate reads the base fresh and rewrites the stored limit, so the state self-heals on the next operate(). admin::update_user_withdrawal_limit already exists and is precisely recommendation Liquidity layer review fixes #2, so the gap is operational, not structural."*

L-04: Zero-supply borrowing turns collectable protocol revenue into zero-utilisation-rate credit exposure

Severity: Low	Impact: Low	Likelihood: Low
Files: packages/liquidity/sources/state/reserve.move	Status: Acknowledged	

Description:

`calc_utilization` deliberately returns zero when total supply is zero to avoid division by zero:

```
if (total_scaled_supply == 0) {
    return 0
};
```

The same result is therefore used for two materially different states: an empty reserve with no debt and a reserve with no supply but nonzero debt. The latter can arise because the vault may retain protocol revenue after all supplier positions have been closed.

A configured with-interest borrower can borrow this retained revenue up to its debt ceiling and the available vault balance. Although the amount remains recorded as debt, it is no longer available for immediate revenue collection and becomes dependent on the borrower's repayment. While supply remains zero, the reserve continues recording `0%` utilization and charging `rateAtUtilizationZero`, regardless of the outstanding debt.

For example:

```
Initial:      supply 0, borrow 0, revenue 100, rate 4%
Borrow 100:   supply 0, borrow 100, utilization 0%, rate 4%
Deposit 10:   utilization becomes 1,000% and the deposit aborts
Deposit 20:   utilization becomes 500% and the deposit succeeds
Afterwards:   existing debt is repriced from 4% at 500% utilization
```

This state makes reserve recovery discontinuous:

1. Deposits that are too small relative to the outstanding debt abort because utilization exceeds the `65,535` encoding limit, preventing gradual recapitalization.

2. The first deposit large enough to succeed can make utilization jump immediately from the previously recorded `0%` to several hundred percent. This prospectively reprices all existing with-interest debt from `rateAtUtilizationZero` to an extreme-utilization rate, potentially reaching the applicable rate cap in a single operation.

Recommendations:

Consider rejecting a positive borrow when the resulting state would have zero total supply and nonzero total borrow. The restriction could apply only to new borrowing so that repayments and deposits remain available for recovering an existing zero-supply state.

If lending protocol-owned revenue without supplier liquidity is intended, consider representing that revenue explicitly as reserve funding and incorporating it into utilization and rate calculations, instead of assigning the resulting debt `0%` utilization solely because supplier supply is zero.

Customer Response:

Acknowledged - *"Borrowing requires an active, governance-authorized UserCap, so this exposure can only land on an already-vetted protocol, not an arbitrary caller. Deferred rather than fixed — rejecting zero-supply-with-debt outright would also block the legitimate case: a WI protocol's governance intentionally funding and drawing down its own market."*

Informational Severity Issues

I-01: Missing event emission for TokenReserve initialization

Files:

packages/liquidity/sources/module/admin.move

Description:

Both `init_token_reserve<T>` and `init_token_reserve_with_metadata<T>` create and share a new `TokenReserve<T>` without emitting a corresponding event. Consequently, event-based indexers and monitoring systems cannot directly detect when a new token reserve is initialised and must instead inspect transaction object changes or already know the token type to derive its reserve address.

Recommendations:

Emit a common token-reserve initialization event from both functions. Consider including the token `TypeName`, reserve object ID, token decimals, and initializer address.

Customer Response:

Fixed in commit 7e470fb.

Fix Review:

Fixed.

I-02: Revenue collection leaves the reserve in an inconsistent update state

Files:

packages/liquidity/sources/module/admin.move
packages/liquidity/sources/state/reserve.move

Description:

`collect_revenue` calls `update_exchange_price` before calculating the available revenue:

```
let (supply_exchange_price, borrow_exchange_price) =
    token_reserve.update_exchange_price(now);
```

The function stores the current supply and borrow exchange prices and advances `last_update_timestamp`, but it does not recalculate `last_utilization` or `borrow_rate`.

After revenue collection, the reserve contains:

```
Supply exchange price:   updated at T1
Borrow exchange price:   updated at T1
Last update timestamp:   updated to T1
Last utilization:        calculated during the previous full update
Borrow rate:             calculated during the previous full update
```

`update_exchange_price` also emits `LogUpdateExchangePrices` containing the new exchange prices together with the previously stored utilization and borrow rate. The stored fields and emitted event therefore represent an inconsistent snapshot: the utilization implied by the current exchange prices may differ from `last_utilization`, and the corresponding rate may differ from `borrow_rate`.

Under the intended piecewise-constant rate model, the stored borrow rate remains active until `update_exchange_prices_and_rates` performs a full update. Revenue collection therefore splits accrual into two intervals using the same rate and this does not by itself demonstrate incorrect accrual. It nevertheless introduces an additional compounding and rounding point, so the exchange prices at a later update may differ from a path without revenue collection.

This inconsistent snapshot can confuse integrations and indexers that assume the stored fields (or `LogUpdateExchangePrices`) represent values calculated from the same reserve state. The

current Solana implementation follows the same behavior, while regular permissionless updates limit how long the inconsistent state persists.

Recommendations:

Document that `collect_revenue` advances the exchange prices and timestamp while retaining the previous utilization and borrow rate, including the resulting inconsistent snapshot and additional compounding and rounding point. If all reserve fields should represent the same update state, use `update_exchange_prices_and_rates` during revenue collection.

Customer Response:

Acknowledged - *"update_exchange_price advances prices and last_update_timestamp but not last_utilization / borrow_rate. Under the intended piecewise-constant rate model both intervals accrue at the same stored rate, so accrual is correct; the cost is one extra compounding/rounding point and a LogUpdateExchangePrices payload that mixes new prices with the previous utilization. Solana behaves identically, and permissionless updates bound how long it persists."*

I-03: Subtracting the floored interest-free share rounds supplier yield up

Files:

packages/liquidity/sources/state/reserve.move

Description:

When with-interest (WI) borrow exceeds interest-free (IF) borrow, the share of borrowers paying interest is calculated by subtracting the floored IF share:

```
rate_output -
    (borrow_ratio * rate_output / (four_decimals + borrow_ratio))
```

As a result, the WI share is effectively rounded up. The opposite branch directly rounds the WI share down.

The impact is minor because the discrepancy is only one unit at `1e17` intermediate precision and is generally removed by subsequent flooring. Additionally, the intended deployment uses only WI borrows. With no IF borrow, `borrow_ratio` is zero and the calculation returns exactly `rate_output`, so this discrepancy is not encountered. The same calculation is also used by the EVM and Solana implementations.

Recommendations:

Consider ceiling the deducted term so the resulting WI share is rounded down:

```
rate_output - math::div_ceil_u128(
    borrow_ratio * rate_output,
    four_decimals + borrow_ratio,
)
```

If preserving reference parity is preferred, consider documenting the rounding behavior and the assumption that IF borrowing will not be enabled.

Customer Response:

Fixed in commit 7e470fb.

Fix Review:

Fixed by documentation.

I-04: Utilization checks reuse the max_rate constant

Files:

packages/liquidity/sources/math/rate_model.move
packages/liquidity/sources/state/reserve.move

Description:

The Liquidity Layer uses `constants::max_rate()` when validating utilization in two locations:

- `reserve::calc_utilization`, before storing the calculated utilization.
- `rate_model.move`, before using utilization during borrow-rate interpolation.

The checks are numerically correct because `max_rate()` equals `65_535`, and both borrow rate and utilization use the same `u16`-equivalent storage domain inherited from Solana. However, `max_rate()` semantically describes the borrow-rate bound, while these checks enforce the maximum representable utilization.

This is also distinct from the configured `max_utilization`, which is an economic restriction applied only to new borrowing.

Recommendations:

Consider defining a dedicated constant such as:

```
public(package) fun max_storable_utilization(): u64 {
    65_535
}
```

Alternatively, derive both limits from a neutral `max_u16()` constant while retaining separate rate and utilization accessors.

Customer Response:

Acknowledged - *"Certora's own note confirms the checks are numerically correct (borrow rate and utilization share the same u16-equivalent 65,535 bound inherited from Solana); max_rate() is just reused across two semantically distinct checks. A dedicated max_storable_utilization() constant would only improve naming clarity, not change behavior."*

I-05: Expansion parameters are applied retrospectively after config updates

Files:

packages/liquidity/sources/module/admin.move

Description:

When updating an existing supply or borrow configuration, the implementation preserves the stored dynamic limit and `last_update`, including during mode changes:

```
position.set_borrow_config(user_borrow_config);
// debt_ceiling and last_update remain unchanged
```

Suppose an operation at `t0` stores the current limit and timestamp, an administrator changes the expansion parameters at `t1`, and the next operation occurs at `t2`. That operation calculates:

```
let time_elapsed = now - last_update;
```

Consequently, it uses the new `expand_percent` and `expand_duration` over the entire `t2 - t0` interval, including the period before the new configuration existed. The resulting expansion may be partial or may immediately reach the fully expanded limit when the elapsed time is sufficiently large.

This occurs on both sides:

- Supply: the withdrawal limit may immediately decrease, increasing the withdrawable amount.
- Borrow: the debt ceiling may immediately increase, up to the fully expanded ceiling permitted by the new configuration.

The behavior matches the EVM implementation, whose storage masks preserve the last-process timestamp, and the Solana implementation, whose configuration updates likewise do not refresh it. The Sui comments also explicitly acknowledge that `last_update` remains untouched. Therefore, this is primarily an operational consideration.

Recommendations:

If configuration changes are intended to apply prospectively, calculate the effective limit at the update timestamp using the previous configuration, reconcile it with the new constraints, and

store `last_update = now`. Supply decay should also be handled consistently during this settlement.

Otherwise, clearly document that expansion parameter changes apply retrospectively and account for this when preparing configuration updates.

Customer Response:

Fixed in commit 7e470fb & a79be0e.

Fix Review:

Fixed by documentation.

I-06: Withdrawals can push utilization above max_utilization

Files:

packages/liquidity/sources/module/user.move

Description:

The `max_utilization` property is strictly validated only when explicit borrow actions occur. A malicious/misbehaving subprotocol can inflate utilization via a PTB encompassing successive operations (e.g. deposit + borrow followed by withdrawal) bypassing the cap entirely. Existing borrows keep accruing and never re-check the limit until someone tries to borrow again.

The utilization cap is documented as a risk control against over-borrowing relative to supply, but the implementation does not guarantee "utilization never exceeds X." It only guarantees "no additional borrow once util > X." Integrators may still treat `max_utilization` as a hard market ceiling.

Recommendations:

Enforce `utilization <= max_utilization` on every operate that raises utilization, including withdraw-only paths.

Customer Response:

Acknowledged - *"Deliberate: the cap is a borrow-side risk control, not a solvency bound on total utilization. A withdrawal returns principal already owed to the supplier; gating it on utilization would withhold a user's own funds because of how much other protocols have borrowed. A borrow is new exposure the protocol is choosing to extend, which is exactly what the cap exists to gate — so it's enforced as "no additional borrow once utilization > X", not "utilization can never exceed X", matching EVM and Solana. Enforcing it on withdraw-only paths would also let one protocol's borrowing block another protocol's exits out of shared liquidity, a materially worse failure mode. The documentation should state the guarantee precisely rather than the implementation changing."*

I-07: Borrow mode-switch conversion can leave amount > debt_ceiling, violating the borrow-limit invariant

Files:

packages/liquidity/sources/module/admin.move

Description:

In `update_user_borrow_configs`, the mode-change branch converts the position amount and the debt ceiling at the same settled `borrow_exchange_price`, but with opposing rounding: the amount is converted with `mul_div_ceil_u128` while the debt ceiling is converted with `mul_div_u128`. Because `ceil(a)` can exceed `floor(b)` even when the pre-switch position satisfies `amount <= debt_ceiling`, a mode switch can leave the stored position with outstanding debt strictly above its stored debt ceiling.

Recommendations:

Convert `debt_ceiling` with the same rounding as amount (ceil it too), so the stored ceiling cannot land below the converted debt.

Customer Response:

Acknowledged - *"Auth-only, one rounding unit of overshoot, and it self-corrects on the next operation's ceiling recomputation."*

I-08: Dust ratio early return accrues borrow EP while freezing supply EP, diverting essentially all WI borrower interest to revenue

Files:
packages/liquidity/sources/state/reserve.move

Description:

In `calculate_exchange_prices`, the borrow exchange price is accrued first. Then, if with-interest supply is smaller than interest-free supply and the floored `supply_ratio` is `0` (i.e. WI is below 0.01% of IF: `WI · 1e4 < IF`), the function returns early with the new borrow EP and the unchanged supply EP.

While that dust-WI state lasts:

- WI borrowers keep paying full interest (borrow EP grows)
- WI suppliers get no SEP growth (`fee` never applied)
- The whole WI interest stream becomes vault surplus → `calc_revenue` / `collect_revenue`

So an IF-heavy market with a tiny WI residual behaves like an unintended ~100% fee. When WI later rises enough for `supply_ratio >= 1`, there is no catch-up for the skipped windows.

Recommendations:

When `WI < IF`, compute `supply_ratio` in `get_with_interest_vs_free_ratio` at higher precision (e.g. `1e10` instead of `1e4`) so dust WI does not floor to `0`. In the matching `ratio_supply_yield` branch (`supply_with_interest_normal < total_supply_interest_free`), scale that ratio back down so the rest of accrual stays on the existing `1e17` / `1e4` scale.

Customer Response:

Fixed in commit 7e470fb.

Fix Review:

Fixed by documentation.

I-09: UserCaps can be minted for unlisted tokens

Files:

packages/liquidity/sources/module/admin.move

Description:

`mint_user_cap<T>` does not verify that a `TokenReserve<T>` has been created. Governance can therefore mint and activate a capability for an unsupported token:

```
public fun mint_user_cap<T>(
    auth: &Auth,
    registry: &mut Registry,
    user: Holder,
    ctx: &TxContext,
): UserCap<T> {
    auth::assert_governance(registry, auth, ctx);
    user_cap::mint<T>(registry, user)
}
```

Although the cap cannot be used without a corresponding reserve, its deterministic `(Holder, T)` address becomes permanently claimed and cannot be minted again. An accidental mint for an incorrect token type may therefore create an unnecessary or unrecoverable capability state.

Recommendations:

Require the reserve to exist before minting:

```
assert!(
    registry.get_reserve_exists<T>(),
    errors::registry_reserve_not_found(),
);
```

Customer Response:

Fixed in commit 7e470fb.

Fix Review:

Fixed.

I-10: Rounded mixed-mode supply ratios can over-credit WI suppliers

Files:

packages/liquidity/sources/state/reserve.move

Description:

In mixed-mode markets, borrower interest is distributed only among WI suppliers, so the calculation increases their yield based on how small WI supply is relative to total supply. When WI supply is smaller than IF supply, `calculate_exchange_prices` calculates `floor(WI * 10,000 / IF)` and then inverts the rounded result. This turns a small rounding loss into a large overestimate: a ratio of `1.99` becomes `1`, so the code divides by `1` instead of `1.99` and nearly doubles supplier yield.

For example, consider:

```
WI supply:    199
IF supply:    1,000,000
WI borrow:    800,000
Borrow rate:  5%
Fee:          5%
Duration:     1 year
```

The ratio becomes:

```
floor(199 * 10,000 / 1,000,000) = 1
```

With utilization `7,998`, the calculation increases SEP from `1e12` to approximately `380.943e12`. The 199 WI suppliers are credited approximately `75,608.66` tokens of interest, although WI borrowers paid only `40,000`; this creates approximately `35,608.66` tokens of unbacked supplier claims, even before accounting for the protocol's `2,000` token fee.

This requires mixed-mode markets with IF supply (`mode = 0`). Since mode `0` is not currently planned for use, the issue is informational. The same calculation is present in the EVM and Solana implementations.

Recommendations:

Document this behavior and revisit the calculation if mode `0` is ever planned. A mixed-mode implementation could calculate `IF / WI` or `total supply / WI` directly at full precision instead of inverting a previously rounded ratio.

Customer Response:

Fixed in commit 7e470fb.

Fix Review:

Fixed.

I-11: Zero object IDs can be registered as valid holders

Files:

packages/liquidity/sources/module/admin.move
packages/liquidity/sources/utils/keys.move

Description:

`holder_object` accepts any `ID`, and `mint_user_cap` does not reject `Holder::Object(0x0)`. Governance can therefore mint and activate a deterministic `UserCap` for a holder that cannot correspond to a real Sui object.

Once minted, this holder is treated like any other active holder across authorization records, configurations, events, and position mappings. This can create misleading registry state and cause positions to be configured under an invalid object identifier.

As a related observation, the comment stating that naming an object-based account "requires holding it" is not enforced by the liquidity package. `Holder::Object` accepts a bare ID, while authorization is provided separately by `UserCap`; the comment should describe this as an intended usage convention rather than an enforced property.

```
/// Name of a protocol account.
public enum Holder has copy, drop, store {
    /// Caller running SEVERAL accounts: one object per market (a vault per pair,
a DEX per
    /// pool). Passing `object::id(self)` means naming an account requires
holding it.
    Object(ID),
    /// ...
}
```

Recommendations:

Reject `Holder::Object(0x0)` through a shared holder-validation function called during minting and activation. Also update the `Holder::Object` comment to clarify that the package records an object ID as the account namespace but does not verify possession of the referenced object.

Customer Response:

Fixed in commit 7e470fb.

Fix Review:

Fixed.

I-12: Single-admin key is total governance — no quorum or threshold anywhere

Files:

packages/liquidity/sources/module/admin.move

Description:

Every governance action — add/remove admins, grant/revoke auth and guardian roles, redirect the revenue collector, bump the package version, mint caps for arbitrary holders — is gated only on possession of a single `AdminRole` record in the shared `Auth` object, and any one admin acts alone. One compromised admin key (or one malicious admin) can unilaterally grant itself auth/guardian, add a co-admin, redirect revenue, mint a cap naming any victim holder, and migrate the version — total protocol compromise with no threshold or quorum anywhere.

Recommendations:

Require a quorum for sensitive governance actions.

Customer Response:

Acknowledged - *"Deliberate: the quorum lives at the key layer (a multisig / threshold signer holding the admin role), not in the package. Adding on-chain quorum to every governance action is a governance-model change."*

I-13: Global pause is auth-controlled in both directions, so one compromised auth can freeze or unilaterally unfreeze all operate legs

Files:

packages/liquidity/sources/module/admin.move

Description:

`set_global_pause` uses `assert_auth` for both pause and unpause, unlike token/user pauses where guardians pause and governance unpauses. A single compromised auth address can halt supply/withdraw/borrow/payback globally for every protocol, or clear a freeze that governance intended to keep in place during an incident, without any governance signature.

Recommendations:

Align global pause with token/user scope: auth or guardian may pause; only governance may unpause

Customer Response:

Acknowledged - *"set_global_pause intentionally matches the existing EVM (changeStatus) and Solana (change_status) design, where the global kill switch is auth-controlled in both directions. This shape is already accepted conditioned on auth being a dedicated multisig, not an EOA the same will hold on Sui. We're keeping this consistent across all three chains rather than introducing a Sui-specific divergence."*

I-14: Sub-minimum residual positions cannot be settled through ordinary single-leg operations

Files:

packages/liquidity/sources/module/user.move

Description:

`operate_internal` rejects an operation when both leg values are below `min_operate_amount` (`10`). At the same time, supply withdrawals and borrow repayments abort when the converted amount exceeds the remaining position rather than capping it.

Consequently, a position cannot be closed through an ordinary single-leg operation when the normal-token amount required to settle its remaining balance is below `10`:

- Submitting the exact settlement amount is rejected as below the minimum.
- Submitting `10` may convert to more than the remaining raw balance and abort as an over-settlement.

This applies to with-interest positions after converting the requested normal-token amount into raw position units using the current exchange price. Exchange-price accrual may eventually raise the normal value of some residual positions above `10`, but very small positions can remain unsettled for a prolonged or practically indefinite period.

A holder may sometimes work around the restriction by including another operation leg of at least `10`. This is not universally available because the other position may be unconfigured, paused, or constrained by its borrow or withdrawal limit.

Small residual balances may therefore remain recorded in user positions and aggregate supply or borrow totals even though the holder intends to close the position. The affected value is bounded by the minimum-operation threshold, making this primarily a position-cleanup and accounting-liveness issue.

Recommendations:

Permit an active leg below `min_operate_amount` when applying the existing exchange-price conversion and rounding logic would settle that position exactly to zero.

The exception should be evaluated after processing the leg's normal conversion logic:

- If the resulting supply or borrow balance is zero, allow the sub-minimum operation.
- If any balance remains, continue enforcing `min_operate_amount`.

The minimum should be enforced independently for each active leg so that an unrelated leg of at least `10` cannot authorize a sub-minimum partial operation.

Customer Response:

Acknowledged - *"Bounded by min_operate_amount (10 raw units), a position-cleanup and accounting-liveness issue rather than a loss. The proposed exception (allow a sub-minimum leg when conversion settles the position exactly to zero, enforced per-leg) is correct but touches the hottest guard in operate_internal; deferred to its own change with its own test pass."*

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.