OtterSec

Fluid Liquidity Layer

Security Assessment

Prepared for
Fluid

Prepared by
Otter Audits, LLC

Prepared on
August 26th, 2026

Andreas Mantzoutas
andreas @osec.io
Thiago Tavares
thitav @osec.io

Table of Contents

1. Executive Summary 2
1.1. Overview 2
1.2. Key Findings 2
2. Scope 3
3. Findings 4
3.1. Findings Summary 4
4. Advisories 5
OS-FLL-ADV-00 Retroactive Borrow Accrual Overstates Supply Yield 6
OS-FLL-ADV-01 Inconsistent Access Control for Pause Operations 8
OS-FLL-ADV-02 Auth can Reauthorize Governance-Revoked Users 11
OS-FLL-ADV-03 Incorrect Rounding of Interest Ratios 13
OS-FLL-ADV-04 Revenue Collection Leaves Interest Rate Stale 16
OS-FLL-ADV-05 Base Debt Ceiling Is Not Immediate 18
5. Suggestions 19
OS-FLL-SUG-00 Ambiguous Destinations in Operation Events 20
OS-FLL-SUG-01 Redundant Interest Amount Cap Check 22
OS-FLL-SUG-02 Constant Values Exposed Through Functions 24
OS-FLL-SUG-03 Narrow Exchange-Price Storage Type 25
A. Vulnerability Rating Scale 26
B. Procedure 27

© 2026 Otter Audits LLC. All Rights Reserved.

1. Executive Summary

1.1. Overview

Fluid engaged OtterSec to conduct a security assessment of the `liquidity-layer` program. This assessment was conducted between August 4th and August 14th, 2026. For information on our auditing methodology, refer to Appendix B.

1.2. Key Findings

We produced 10 findings throughout this audit engagement.

Among the most impactful issues, we found that exchange-price calculation combines opening utilization and borrow-rate values with an end-of-period borrower ratio. This may retroactively overstate supplier yield, over-crediting suppliers relative to accrued borrower debt and creating a reserve deficit (● OS-FLL-ADV-00). Another issue concerns flooring WI-to-IF ratios before inverting or complementing them, which introduces an upward rounding bias that may overstate supplier yield. The supply-side error may be severe enough to over-credit suppliers and create an accounting deficit (● OS-FLL-ADV-03).

Also, pause authorization varies by scope, allowing guardians to pause user-level actions but not reserve-wide actions. This inconsistent privilege model expands guardian authority and complicates enforcement of the intended action-based policy (● OS-FLL-ADV-01). Additionally, an Auth holder can reactivate a user whose capability was revoked by governance, allowing the existing cap to pass admission checks again and enabling a lower-tier role to override governance's access-control decision (● OS-FLL-ADV-02).

Furthermore, the revenue collection function updates exchange prices and the timestamp without refreshing utilization or the borrow rate. Subsequent interest may therefore accrue using a stale rate until the reserve is fully updated (● OS-FLL-ADV-04).

We also highlighted an ambiguous event emission that may lead off-chain consumers to draw incorrect conclusions (● OS-FLL-SUG-00). We further suggested removing a redundant validation check (● OS-FLL-SUG-01) and replacing constant functions with macros to improve gas efficiency (● OS-FLL-SUG-02). Lastly, we recommended using the `u128` type for the exchange prices (● OS-FLL-SUG-03).

© 2026 Otter Audits LLC. All Rights Reserved.

2. Scope

Fluid's `liquidity-layer` is the shared asset-management and accounting foundation for the wider Fluid protocol on Sui. It is designed to hold liquidity for each supported coin and allow authorized Fluid applications to supply assets, withdraw them, borrow liquidity, and repay debt through a common reserve.

The source code was delivered to us in a Git repository at `https://github.com/Instadapp/fluid-packages-sui`.

This audit was performed against commit `8e7da4a`.

© 2026 Otter Audits LLC. All Rights Reserved.

3. Findings

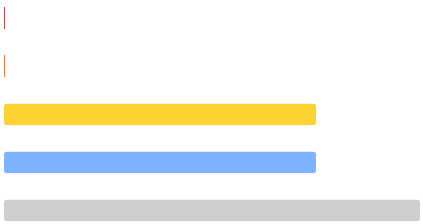
Findings were rated using the criteria in Appendix A.

We split the findings into **advisories** and **suggestions**. Advisories have an immediate impact and should be remediated as soon as possible. Suggestions do not have an immediate impact but will aid in mitigating future vulnerabilities.

Items are tracked with a stable identifier: `OS-FLL-ADV-*` for advisories and `OS-FLL-SUG-*` for suggestions.

3.1. Findings Summary

Severity	Count
Critical	0
High	0
Medium	3
Low	3
Informational	4



© 2026 Otter Audits LLC. All Rights Reserved.

4. Advisories

Here, we present a technical analysis of the 6 vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

ID	Severity	Status	Description
OS-FLL-ADV-00	Medium	Resolved	Retroactive Borrow Accrual Overstates Supply Yield
OS-FLL-ADV-01	Medium	Resolved	Inconsistent Access Control for Pause Operations
OS-FLL-ADV-02	Medium	Resolved	Auth can Reauthorize Governance-Revoked Users
OS-FLL-ADV-03	Low	Resolved	Incorrect Rounding of Interest Ratios
OS-FLL-ADV-04	Low	Ack'd	Revenue Collection Leaves Interest Rate Stale
OS-FLL-ADV-05	Low	Resolved	Base Debt Ceiling Is Not Immediate

© 2026 Otter Audits LLC. All Rights Reserved.

Retroactive Borrow Accrual Overstates Supply Yield

ID	Severity	Status	Found in
OS-FLL-ADV-00	Medium	Resolved	8d9d13e

Description

`reserve::calculate_exchange_prices` derives both exchange prices for the elapsed interval from the reserve's stored rate and utilization state. The supply-side calculation therefore relies on a utilization value captured at the beginning of the interval while determining supplier yield for that same interval. The function first accrues `borrow_exchange_price` across the full elapsed interval, which produces an end-of-period debt exchange price before the supplier-yield inputs are derived.

packages/liquidity/sources/state/reserve.move

```
public(package) fun calculate_exchange_prices<T>(
    self: &TokenReserve<T>,
    now: TimestampS,
): (u128, u128) {
    [...]
    // calculate new borrow exchange price.
    // formula borrowExchangePriceIncrease: previous price * borrow rate *
    secondsSinceLastUpdate_.
    borrow_exchange_price =
        borrow_exchange_price + math::div_ceil_u128(
        borrow_exchange_price * borrow_rate * seconds_since_last_update,
        seconds_per_year * four_decimals,
    );
    [...]
}
```

It then utilizes that already-accrued price to normalize the interest-bearing borrow total to calculate `borrow_with_interest_normal` and consequently, the resulting ratio to determine the proportion of borrowers paying interest. This end-of-period borrower proportion is combined with the stored opening utilization and borrow rate to calculate `supply_exchange_price` growth for the same interval.

packages/liquidity/sources/state/reserve.move

```
public(package) fun calculate_exchange_prices<T>(
    self: &TokenReserve<T>,
    now: TimestampS,
): (u128, u128) {
    [...]
    // Fresh ratio from exact totals, normalized first:
```

© 2026 Otter Audits LLC. All Rights Reserved.

```
        // `get_borrow_with_interest_normal(borrow_exchange_price)` = floor(raw * bep /
        1e12).
        // Uses the post-accrual `borrow_exchange_price` (already reassigned above) —
        matches
        // Solana, which ratios the borrow side after its own local `borrow_exchange_price`
        was
        // already updated.
        // Overflow: same bound as `get_total_borrow` (raw < 2^64, price < 2^64 for
        // storage-sourced inputs -> product < 2^128).
        let borrow_with_interest_normal =
            (self.total_borrow_with_interest as u128) * borrow_exchange_price
                / constants::exchange_prices_precision();
        let borrow_ratio = liquidity_calcs::get_with_interest_vs_free_ratio(
            borrow_with_interest_normal,
            self.total_borrow_interest_free as u128,
        );
        [...]
    }
```

However, since the increased borrower yield is applied retroactively to the entire supply-accrual interval, the supply exchange price may grow by more than the interest accrued to borrowers for that interval. Therefore, this mismatch can over-credit suppliers relative to borrower debt and may cause reserve insolvency.

Remediation

When calculating supply exchange-price growth for an elapsed interval, derive the interest-bearing borrower proportion from the opening exchange price and opening reserve totals, rather than from the borrow exchange price after it has been accrued.

Patch

Resolved in 7e470fb.

© 2026 Otter Audits LLC. All Rights Reserved.

Inconsistent Access Control for Pause Operations

ID	Severity	Status	Found in
OS-FLL-ADV-01	Medium	Resolved	8d9d13e

Description

`pause_token` applies different role checks based on the action: guardian authorization is required for `Borrow` and `Withdraw`, while auth-tier authorization is required for `Supply` and `Payback`. The pause is then recorded on the token reserve, affecting that action for the reserve.

packages/liquidity/sources/module/admin.move

```
698 public fun pause_token<T>(
699     auth: &Auth,
700     registry: &Registry,
701     token_reserve: &mut TokenReserve<T>,
702     action: Action,
703     ctx: &TxContext,
704 ) {
705     // Guardian may freeze the outflow actions (stolen-guardian-key blast radius);
    Supply/
706     // Payback token-wide pause needs Auth tier.
707     if (action == pause::action_borrow() || action == pause::action_withdraw()) {
708         auth::assert_guardian(registry, auth, ctx);
709     } else {
710         auth::assert_auth(registry, auth, ctx);
711     };
712     // pause requires not-already-paused for this action -> else aborts.
713     assert!(
714         pause::get_allows(&token_reserve.get_pause_flags(), action),
715         errors::admin_module_invalid_params(),
716     );
717     token_reserve.set_reserve_pause(action, true);
718
719     events::emit_log_token_pause_changed(
720         type_name::with_defining_ids<T>(),
721         action,
722         true,
723         ctx.sender(),
724     );
725 }
```

However, `pause_user` instead performs only a guardian authorization check before selecting either the user's supply-side or borrow-side position and setting the requested action's pause state. Consequently, a guardian may pause all four action types at user scope, even though the same role cannot pause `Supply` or `Payback` at reserve scope.

© 2026 Otter Audits LLC. All Rights Reserved.

packages/liquidity/sources/module/admin.move

```
750 public fun pause_user<T>(
751     auth: &Auth,
752     registry: &Registry,
753     token_reserve: &mut TokenReserve<T>,
754     user: Holder,
755     action: Action,
756     ctx: &TxContext,
757 ) {
758     auth::assert_guardian(registry, auth, ctx);
759 
760     let is_supply_side = action == pause::action_supply() || action ==
    pause::action_withdraw();
761     if (is_supply_side) {
762         // no config set -> aborts.
763         assert!(
764             token_reserve.get_supply_position_exists(user),
765             errors::admin_module_user_not_defined(),
766         );
767         let position = token_reserve.get_supply_position_mut(user);
768         assert!(!position.get_supply_configs_not_set(),
        errors::admin_module_user_not_defined());
769         // pause requires not-already-paused for this action -> else aborts.
770         assert!(!position.get_supply_pause(action),
        errors::admin_module_invalid_params());
771         position.set_supply_pause(action, true);
772     } else {
773         // no config set -> aborts.
774         assert!(
775             token_reserve.get_borrow_position_exists(user),
776             errors::admin_module_user_not_defined(),
777         );
778         let position = token_reserve.get_borrow_position_mut(user);
779         assert!(!position.get_borrow_configs_not_set(),
        errors::admin_module_user_not_defined());
780         // pause requires not-already-paused for this action -> else aborts.
781         assert!(!position.get_borrow_pause(action),
        errors::admin_module_invalid_params());
782         position.set_borrow_pause(action, true);
783     };
784     [...]
785 }
```

© 2026 Otter Audits LLC. All Rights Reserved.

These differing checks create inconsistent pause authority between reserve-level and user-level controls that gate the same operation actions. The global pause setter is also auth-tier only, so the current role boundaries vary by pause scope rather than following a single action-based policy.

Remediation

Align `pause_token` and `pause_user` to one action-based authorization policy. Require auth-tier authorization for `Payback` and `Withdraw` pauses, and permit either auth-tier or guardian authorization for `Supply` and `Borrow` pauses.

Patch

Resolved in 7f3b159.

© 2026 Otter Audits LLC. All Rights Reserved.

Auth can Reauthorize Governance-Revoked Users

ID	Severity	Status	Found in
OS-FLL-ADV-02	Medium	Resolved	8d9d13e

Description

`admin::set_user_authorization` applies different authorization tiers depending on the requested state: an `Auth` holder may activate a user, while only governance may revoke one. Consequently, an `Auth` holder may reactivate an existing user-cap record after governance has deliberately disabled it, overriding that governance decision.

packages/liquidity/sources/module/admin.move

```
138 public fun set_user_authorization<T>(
139     auth: &Auth,
140     registry: &mut Registry,
141     user: Holder,
142     on: bool,
143     ctx: &TxContext,
144 ) {
145     if (on) {
146         auth::assert_auth(registry, auth, ctx);
147         user_cap::activate<T>(registry, user);
148     } else {
149         auth::assert_governance(registry, auth, ctx);
150         user_cap::revoke<T>(registry, user);
151     };
152     events::emit_log_set_user_authorization(
153         user_cap::get_user_cap_id<T>(registry, user),
154         user,
155         type_name::with_defining_ids<T>(),
156         on,
157     );
158 }
```

User-cap minting creates the cap and immediately adds its record to the registry's active-cap set. Revocation removes that active record, while activation re-adds the record for an existing cap. Therefore, a revoked user that still holds its capability may again satisfy the active-cap admission check used by user operations once an `Auth` holder reactivates it. This weakens governance's exclusive control over restoring access for previously revoked users.

packages/liquidity/sources/state/user_cap.move

```
38 public(package) fun mint<T>(registry: &mut Registry, holder: Holder): UserCap<T> {
39     let id = derived_object::claim(registry.get_uid_mut(),
       keys::user_cap_key<T>(holder));
```

© 2026 Otter Audits LLC. All Rights Reserved.

```
40      let cap = UserCap<T> { id, holder };
41      registry
42          .active_caps_mut()
43          .add(
44              object::id(&cap),
45              structs::new_cap_record(type_name::with_defining_ids<T>(), holder),
46          );
47      cap
48  }
```

Remediation

Require governance authorization before activating a user in `set_user_authorization`, matching the authorization requirement for revocation.

Patch

Resolved in 7e470fb.

© 2026 Otter Audits LLC. All Rights Reserved.

Incorrect Rounding of Interest Ratios

ID	Severity	Status	Found in
OS-FLL-ADV-03	Low	Resolved	8d9d13e

Description

`liquidity_calcs::get_with_interest_vs_free_ratio` computes the smaller normalized balance as a `1e4`-scaled ratio of the larger balance and uses integer division in both unequal-balance branches. The returned value is therefore always rounded down before its later consumers apply an inversion or complement.

packages/liquidity/sources/math/liquidity_calcs.move

```
/// Ratio of the smaller side vs the larger side in 1e4 precision.
public(package) fun get_with_interest_vs_free_ratio(
    with_interest: u128,
    interest_free: u128,
): u128 {
    let four_decimals = constants::four_decimals() as u128;
    if (with_interest > interest_free) {
        // ratio with interest being larger. floor.
        interest_free * four_decimals / with_interest
    } else if (with_interest < interest_free) {
        // ratio with interest free being larger. floor.
        with_interest * four_decimals / interest_free
    } else if (with_interest > 0) {
        // amounts match exactly and are not 0 -> ratio 1 (1e4).
        four_decimals
    } else {
        // total = 0.
        0
    }
}
```

When IF supply exceeds WI supply, `calculate_exchange_prices` subsequently inverts this truncated ratio, turning the initial downward rounding into upward rounding. For small non-zero ratios, this error can be substantial: if the exact scaled ratio lies between 1 and 2 but is truncated to 1, its inverse can approach twice the correct value. This overstates `ratio_supply_yield`, causing the supply exchange price to accrue more interest than the interest-bearing borrowers generated and potentially creating an immediate accounting deficit.

© 2026 Otter Audits LLC. All Rights Reserved.

packages/liquidity/sources/state/reserve.move

```
public(package) fun calculate_exchange_prices<T>(
    self: &TokenReserve<T>,
    now: TimestampS,
): (u128, u128) {
    [...]
    let mut ratio_supply_yield = if (
        supply_with_interest_normal < (self.total_supply_interest_free as u128)
    ) {
        // ratio is supplyWithInterest / supplyInterestFree
        // (supplyInterestFree is bigger).
        if (supply_ratio == 0) {
            // supply with interest is tiny vs interest-free
            // (floored ratio 0) -> no one is earning interest -> no supply EP update.
            return (supply_exchange_price, borrow_exchange_price)
        };
        // invert the ratio. floor.
        // Overflow: 1e17 * 1e4 = 1e21 < 2^128 (constant product); supply_ratio >= 1
        here.
        let supply_ratio = rate_output * four_decimals / supply_ratio;
        [...]
    }
    [...]
}
```

A similar directional error occurs when WI borrowing exceeds IF borrowing. The function floors the IF-to-WI ratio and then computes its complement, so subtracting the underestimated component produces an overestimated interest-bearing borrow share. This also increases `ratio_supply_yield` and over-credits interest-bearing suppliers, although the error is bounded by the `1e4` ratio precision and remains below approximately one basis point. Thus, as long as the protocol fee is non-zero, it will be absorbed by the protocol revenue.

packages/liquidity/sources/state/reserve.move

```
public(package) fun calculate_exchange_prices<T>(
    self: &TokenReserve<T>,
    now: TimestampS,
): (u128, u128) {
    [...]
    let borrow_ratio = if (
        borrow_with_interest_normal < (self.total_borrow_interest_free as u128)
    ) { [...] } else {
        // x = 100% - borrowRatio / (100% + borrowRatio). floor
        // then subtraction. Overflow: same 1e21 bound; the subtrahend is <= 5e16 <=
        1e17,
```

© 2026 Otter Audits LLC. All Rights Reserved.

```
11          // so the u128 subtraction can never underflow. Max value 1e17 (100% paying
            yield).
12          rate_output - (borrow_ratio * rate_output / (four_decimals + borrow_ratio))
13      };
14      [...]
15  }
```

Remediation

Do not reuse an always-floored smaller-to-larger ratio where the subsequent calculation inverts or complements it. Compute the required supply and borrow proportions directly from the normalized WI and IF amounts using sufficiently wide `mul_div` operations and an explicit conservative rounding direction.

Patch

Resolved in 7e470fb.

© 2026 Otter Audits LLC. All Rights Reserved.

Revenue Collection Leaves Interest Rate Stale

ID	Severity	Status	Found in
OS-FLL-ADV-04	Low	Ack'd	8d9d13e

Description

`admin::collect_revenue` accrues and persists the current supply and borrow exchange prices before computing and withdrawing revenue, as shown in `collect_revenue`. However, this path calls the price-only update routine and does not recalculate or store the reserve's utilization and borrow rate alongside the new prices.

packages/liquidity/sources/module/admin.move

```
671 public fun collect_revenue<T>(
672     auth: &Auth,
673     registry: &Registry,
674     token_reserve: &mut TokenReserve<T>,
675     clock: &Clock,
676     ctx: &mut TxContext,
677 ) {
678     auth::assert_auth(registry, auth, ctx);
679     let now = units::from_clock(clock);
680 
681     let revenue_collector = registry.get_revenue_collector();
682     assert!(revenue_collector != @0x0,
        errors::admin_module_revenue_collector_not_set());
683 
684     // accrue & persist exchange prices + last_update before computing revenue
685     // consistent with every other entrypoint that touches the reserve.
686     let (supply_exchange_price, borrow_exchange_price) =
        token_reserve.update_exchange_price(now);
687     let revenue_amount = token_reserve.calc_revenue(supply_exchange_price,
        borrow_exchange_price);
688 
689     // transfer the amount to the revenue collector when > 0. Vault insufficiency
        aborts
690     // inside `withdraw` (custody-confined).
691     if (revenue_amount > 0) {
692         let funds = token_reserve.withdraw(revenue_amount);
693         transfer::public_transfer(coin::from_balance(funds, ctx),
            revenue_collector);
694     };
695     events::emit_log_collect_revenue(type_name::with_defining_ids<T>(),
        revenue_amount);
696 }
```

© 2026 Otter Audits LLC. All Rights Reserved.

Utilization is derived from the reserve's scaled supply and borrow totals using the applicable exchange prices. Consequently, when with-interest balances are present and revenue collection changes those prices, the persisted utilization and borrow rate can no longer reflect the newly stored prices. The next exchange-price accrual therefore uses the stale borrow rate for the elapsed period, which may result in incorrect interest accrual until reserve state is subsequently refreshed through a path that recalculates rates.

Remediation

Replace the price-only reserve update in `collect_revenue` with the reserve update path that calculates and persists exchange prices, utilization, and the borrow rate using the same `now` timestamp before revenue is calculated.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Base Debt Ceiling Is Not Immediate

ID	Severity	Status	Found in
OS-FLL-ADV-05	Low	Resolved	8d9d13e

Description

`base_debt_ceiling` is intended to provide immediately available borrowing capacity without waiting for stepped expansion. However, `calc_borrow_limit_before_operate` returns the base ceiling only when the fully expanded ceiling is below it, rather than ensuring the computed current limit is at least the base. As a result, when `current_borrow_limit < base_limit < max_expanded_borrow_limit`, the function returns the lower current limit. Although normal operation preserves the expected invariant, an administrator can create this state by increasing the base ceiling above the previously stored limit. The affected user must then wait for time-based expansion before accessing borrowing capacity that should have been available immediately.

packages/liquidity/sources/math/liquidity_calcs.move

```
public(package) fun calc_borrow_limit_before_operate(
    user_amount: u256,
    last_limit: u128,
    expand_percent: u64,
    expand_duration: u64,
    base_limit: u128,
    max_limit: u128,
    last_update: u64,
    now: u64,
): u128 {
    // max expansion limit = userBorrow * expandPercent / 1e4
    let max_expansion_limit = expansion_amount(user_amount, expand_percent);
    let max_expanded_borrow_limit = user_amount + max_expansion_limit;

    if ((max_expanded_borrow_limit as u128) < base_limit) {
        return base_limit
    };
    [...]
}
```

Remediation

Clamp the calculated borrow limit to at least `base_debt_ceiling`, while continuing to enforce `max_debt_ceiling`.

Patch

Resolved in PR#43.

© 2026 Otter Audits LLC. All Rights Reserved.

5. Suggestions

Here, we present a discussion of the 4 suggestions we identified during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Severity	Status	Description
OS-FLL-SUG-00	Informational	Ack'd	Ambiguous Destinations in Operation Events
OS-FLL-SUG-01	Informational	Ack'd	Redundant Interest Amount Cap Check
OS-FLL-SUG-02	Informational	Ack'd	Constant Values Exposed Through Functions
OS-FLL-SUG-03	Informational	Resolved	Narrow Exchange-Price Storage Type

© 2026 Otter Audits LLC. All Rights Reserved.

Ambiguous Destinations in Operation Events

ID	Severity	Status	Found in
OS-FLL-SUG-00	Informational	Ack'd	8d9d13e

Description

In the current implementation, `events::emit_log_operate` records `withdraw_to` and `borrow_to` in `LogOperate` as they were supplied to the operate function.

packages/liquidity/sources/types/events.move

```
177 public(package) fun emit_log_operate(
178     user: Holder,
179     token: TypeName,
180     supply_amount: u64,
181     supply_is_negative: bool,
182     borrow_amount: u64,
183     borrow_is_negative: bool,
184     withdraw_to: address,
185     borrow_to: address,
186     supply_exchange_price: u64,
187     borrow_exchange_price: u64,
188 ) {
189     event::emit(LogOperate {
190         user,
191         token,
192         supply_amount,
193         supply_is_negative,
194         borrow_amount,
195         borrow_is_negative,
196         withdraw_to,
197         borrow_to,
198         supply_exchange_price,
199         borrow_exchange_price,
200     })
201 }
```

However, there is no guarantee that the transferred funds will reach the supplied addresses. When `TransferType::Skip` is selected, withdrawal and borrow balances are returned to the caller rather than transferred directly to the destination addresses. Because the operation event still records those supplied addresses, off-chain consumers may incorrectly treat them as final recipients even though the event alone does not prove direct delivery.

© 2026 Otter Audits LLC. All Rights Reserved.

Remediation

Include the transfer mode in `LogOperate` and document `withdraw_to` and `borrow_to` as requested destinations when transfers are skipped.

Patch

Instadapp acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Redundant Interest Amount Cap Check

ID	Severity	Status	Found in
OS-FLL-SUG-01	Informational	Ack'd	8d9d13e

Description

`reserve::calc_utilization` derives aggregate scaled supply and borrow amounts by adding each scaled interest-bearing amount to its corresponding scaled interest-free amount, as shown below:

packages/liquidity/sources/state/reserve.move

```
public(package) fun calc_utilization<T>(
    self: &TokenReserve<T>,
    supply_exchange_price: u128,
    borrow_exchange_price: u128,
    supply_amount_positive: bool,
    borrow_amount_positive: bool,
): u64 {
    [...]
    // scale raw with-interest amounts by their exchange price.
    let interest_supply = (self.total_supply_with_interest as u128) *
    supply_exchange_price;
    let interest_borrow = (self.total_borrow_with_interest as u128) *
    borrow_exchange_price;
    [...]
}
```

`check_scaled_amount_overflow` compares both `interest_amount` and `scaled_amount` against the same cap. Because the aggregate scaled amount includes the interest-bearing amount plus a non-negative interest-free component, enforcing the cap on the aggregate amount also bounds the interest-bearing amount.

packages/liquidity/sources/state/reserve.move

```
fun check_scaled_amount_overflow(
    amount_positive: bool,
    interest_amount: u128,
    scaled_amount: u128,
    is_supply: bool,
) {
    if (amount_positive) {
        // Overflow: MAX_TOKEN_AMOUNT_CAP = 2^60 - 1 (~1.15e18) * 1e12 (~2^40) < 2^100
        // < 2^128 — constant product, never overflows.
        let cap_scaled =
            (constants::max_token_amount_cap() as u128) *
            constants::exchange_prices_precision();
```

© 2026 Otter Audits LLC. All Rights Reserved.

```
473             if (interest_amount > cap_scaled || scaled_amount > cap_scaled) {
474                 if (is_supply) {
475                     abort errors::user_module_value_overflow_total_supply()
476                 } else {
477                     abort errors::user_module_value_overflow_total_borrow()
478                 }
479             }
480         }
481     }
```

Consequently, the `interest_amount` comparison is redundant and does not change whether this guard aborts. Removing it would simplify the helper without changing the cap-enforcement behavior for the supply and borrow paths that call it.

Remediation

Remove the `interest_amount` parameter from `check_scaled_amount_overflow`, remove the associated comparison, and update its supply and borrow call sites in `calc_utilization` to pass only the aggregate scaled amount.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Constant Values Exposed Through Functions

ID	Severity	Status	Found in
OS-FLL-SUG-02	Informational	Ack'd	`8d9d13e`

Description

The `fluid_liquidity::constants` module exposes immutable protocol values through zero-argument functions. These accessors provide values used throughout liquidity calculations, reserve state handling, rate calculations, position processing, registry version checks, and user-operation validation. Replacing the immutable constant accessors with Move macros that expand the corresponding typed values at their call sites will reduce any overhead associated with invoking these functions while preserving the existing values and call-site semantics.

Remediation

Replace the zero-argument immutable accessors in `fluid_liquidity::constants` with typed Move macros that expand to the same values.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Narrow Exchange-Price Storage Type

ID	Severity	Status	Found in
OS-FLL-SUG-03	Informational	✓ Resolved	8d9d13e

Description

Currently, `TokenReserve` stores both supply and borrow exchange prices as `u64`, while exchange-price calculations use `u128` working values. The reserve is initialized with a `u64` exchange price, and public getters expose the stored prices as `u64`.

Exchange-price settlement is callable through the public, permissionless `admin::update_exchange_prices` entry point. Before persisting calculated prices, the implementation bounds them to `u64::MAX` and aborts if either price exceeds that limit. As a result, if a price reaches this bound, the affected settlement path will fail rather than silently truncate the value. In the worst-case estimate, the `u64` price limit could be reached after 941 days of daily permissionless updates.

Remediation

Store `supply_exchange_price` and `borrow_exchange_price` as `u128` in `TokenReserve`.

Patch

Resolved in 7e470fb.

© 2026 Otter Audits LLC. All Rights Reserved.

Appendix A.

Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the general findings.

Severity	Description
Critical	Vulnerabilities that immediately result in a loss of availability, consensus divergence, or exploitable memory corruption with minimal preconditions.
High	Vulnerabilities that may result in consensus divergence or denial of service but require specific transaction or network conditions to exploit.
Medium	Vulnerabilities that may result in incorrect behavior or state divergence under specific configurations or edge cases.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.
Informational	Best practices to mitigate future security risks. These are classified as general findings.

Resolution status is one of ✓ Resolved, ○ Ack'd, or × To-do.

© 2026 Otter Audits LLC. All Rights Reserved.

Appendix B.

Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

The design review asks whether the programs's architecture and operating assumptions support its intended behavior. We study the inputs it accepts, the decisions it makes, the information shared between components, and the services or participants it trusts. We then consider how it behaves when those inputs are unexpected or conflicting. The goal is to find conditions that can produce an incorrect decision, inconsistent state, or loss of service.

Design problems often arise from interactions rather than one defective function. Two components may each process a value correctly but interpret it differently. A failure may also leave one component ready to continue while another treats the same operation as incomplete. Local checks can look correct in both cases, so we follow the complete operation across component boundaries.

The implementation review asks whether the code matches the design. We trace untrusted data from its entry point to the point where it affects behavior, checking the assumptions made along the way. The review covers validation, error handling, state updates, concurrency, persistence, resource use, and cleanup. Its exact focus depends on the target's language, runtime, and architecture rather than a fixed list of bug classes.

The boundary between design and implementation is not exact. Missing validation may be a local coding error, while the same omission across several components may reflect an unclear rule. We classify a finding by its root cause, but judge its impact from the resulting behavior and the conditions needed to trigger it.

We begin by reading the target as a team and agreeing on its main components, trust boundaries, and expected behavior. Auditors then review different paths through the system and compare notes where those paths meet. We reproduce suspected issues with focused tests or traces when the setup allows. Before reporting a finding, another auditor reviews its trigger, impact, and proposed remediation.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

© 2026 Otter Audits LLC. All Rights Reserved.