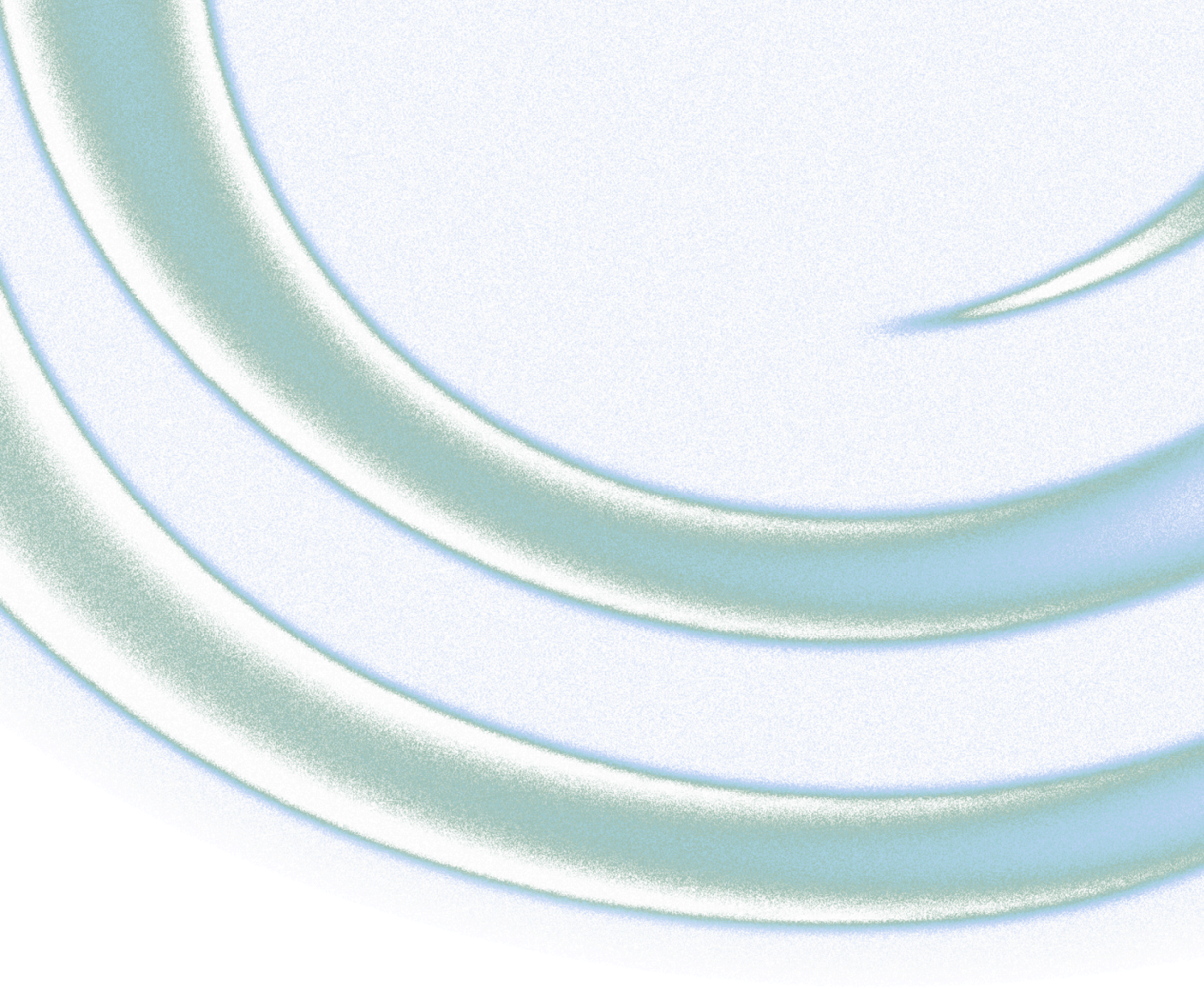


OtterSec

Fluid Vaults

Security Assessment

Prepared for
Fluid

Prepared by
Otter Audits, LLC

Prepared on
October 19th, 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-FLV-ADV-00 Narrowed Exchange Price Checkpoints Block Magnifier Updates 6
OS-FLV-ADV-01 Premature u64 Narrowing Blocks Capped Rebalances 8
OS-FLV-ADV-02 Maximum Withdrawal Desynchronizes Collateral Position Accounting 10
OS-FLV-ADV-03 Missing Vault Asset Compatibility Validation 13
OS-FLV-ADV-04 Collateral-Factor Check Rejects Net Deleveraging Operations 16
OS-FLV-ADV-05 Liquidation-Only Quotes Rejected During Position Improvements 18
OS-FLV-ADV-06 Vault ID Missing from Configuration Update Events 19
OS-FLV-ADV-07 Inconsistent Rounding Weakens the Withdrawal Gap 20
OS-FLV-ADV-08 Inconsistent Collateral Rounding Across Accounting Layers 21
OS-FLV-ADV-09 Placement Dust Inflates Absorbed Debt 22
OS-FLV-ADV-10 Narrow LogAbsorb Fields Abort Large Absorptions 25
5. Suggestions 27
OS-FLV-SUG-00 Low Oracle Prices Reach Unsupported Tick Math 28
OS-FLV-SUG-01 Lack of Functionality to Borrow Stored Display Capability 29
A. Vulnerability Rating Scale 30
B. Procedure 31

© 2026 Otter Audits LLC. All Rights Reserved.

1. Executive Summary

1.1. Overview

Fluid engaged OtterSec to conduct a security assessment of the `vaults` program. This assessment was conducted between September 12th and September 29th, 2026. For information on our auditing methodology, refer to Appendix B.

1.2. Key Findings

We produced 13 findings throughout this audit engagement.

Among the most impactful issues, we identified that exchange prices are narrowed to a smaller integer type when checkpoints are saved, so updates abort once prices grow large (● OS-FLV-ADV-00). Similarly, rebalance differences are narrowed before any cap is applied, so large rebalances abort instead of being reduced (● OS-FLV-ADV-01). Additionally, a maximum collateral withdrawal does not settle for the full collateral value, so the settlement instruction differs from the zero balance recorded for the position (● OS-FLV-ADV-02). Additionally,

Also, vault creation does not fully validate asset decimals, so some configurations that Liquidity cannot support are still accepted (● OS-FLV-ADV-03). Furthermore, the collateral-factor check applies its strictest limit to any withdrawal or borrow, so operations that actually reduce leverage may be rejected (● OS-FLV-ADV-04).

In addition, the safety check always requires both oracle rates, so quotes that only support liquidation are rejected even when an operation improves the position (● OS-FLV-ADV-05). Moreover, the withdrawal-gap check rounds down while Liquidity rounds up, so a withdrawal at the boundary can leave the position one unit short of the intended buffer (● OS-FLV-ADV-07).

We also suggested version-gating the oracle's price functions, since outdated versions remain callable after an upgrade and can return quotes indistinguishable from current ones, bypassing security fixes (● OS-FLV-SUG-00). Lastly, we advised adding functionality to mutably borrow the stored display capability for future metadata updates (● OS-FLV-SUG-01).

© 2026 Otter Audits LLC. All Rights Reserved.

2. Scope

The `vaults` program enables users to deposit collateral and borrow another asset through isolated lending markets. It tracks positions, accrues interest through exchange prices, and enforces oracle-based collateral and liquidation limits. Unsafe positions are efficiently liquidated using a tick-and-branch accounting system, while actual token custody and transfers occur through Fluid Liquidity.

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

This audit was performed against 3f1dc9b and PR#57.

© 2026 Otter Audits LLC. All Rights Reserved.

3. Findings

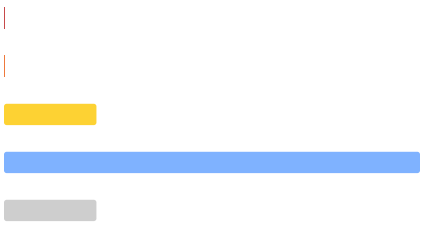
Findings were rated using the criteria in Appendix A.

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-FLV-ADV-*` for advisories and `OS-FLV-SUG-*` for suggestions.

3.1. Findings Summary

Severity	Count
Critical	0
High	0
Medium	2
Low	9
Informational	2



© 2026 Otter Audits LLC. All Rights Reserved.

4. Advisories

Here, we present a technical analysis of the 11 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-FLV-ADV-00	Medium	Ack'd	Narrowed Exchange Price Checkpoints Block Magnifier Updates
OS-FLV-ADV-01	Medium	Ack'd	Premature u64 Narrowing Blocks Capped Rebalances
OS-FLV-ADV-02	Low	Ack'd	Maximum Withdrawal Desynchronizes Collateral Position Accounting
OS-FLV-ADV-03	Low	Ack'd	Missing Vault Asset Compatibility Validation
OS-FLV-ADV-04	Low	Resolved	Collateral-Factor Check Rejects Net Deleveraging Operations
OS-FLV-ADV-05	Low	Resolved	Liquidation-Only Quotes Rejected During Position Improvements
OS-FLV-ADV-06	Low	Resolved	Vault ID Missing from Configuration Update Events
OS-FLV-ADV-07	Low	Ack'd	Inconsistent Rounding Weakens the Withdrawal Gap
OS-FLV-ADV-08	Low	Ack'd	Inconsistent Collateral Rounding Across Accounting Layers
OS-FLV-ADV-09	Low	Ack'd	Placement Dust Inflates Absorbed Debt
OS-FLV-ADV-10	Low	Resolved	Narrow LogAbsorb Fields Abort Large Absorptions

© 2026 Otter Audits LLC. All Rights Reserved.

Narrowed Exchange Price Checkpoints Block Magnifier Updates

ID	Severity	Status
OS-FLV-ADV-00	Medium	Ack'd

Description

The vault calculates liquidity and vault exchange prices as `u128` values in `load_exchange_prices`, but `update_exchange_prices` narrows them when persisting the checkpoint. It casts `liq_supply_ex_price`, `liq_borrow_ex_price`, `vault_supply_ex_price`, and `vault_borrow_ex_price` to `u64`. Consequently, a checkpoint aborts when any calculated or supplied exchange price exceeds the `u64` range, even though the calculation path supports the larger value.

packages/vaults/sources/state/vault_market.move

```
public(package) fun update_exchange_prices<Col, Debt>(
    self: &mut VaultMarket<Col, Debt>,
    liq_supply_ex_price_in: u128,
    liq_borrow_ex_price_in: u128,
    now: TimestampS,
) {
    let (
        liq_supply_ex_price,
        liq_borrow_ex_price,
        vault_supply_ex_price,
        vault_borrow_ex_price,
    ) = self.load_exchange_prices(liq_supply_ex_price_in, liq_borrow_ex_price_in, now);

    self.liquidity_supply_exchange_price = liq_supply_ex_price as u64;
    self.liquidity_borrow_exchange_price = liq_borrow_ex_price as u64;
    self.vault_supply_exchange_price = vault_supply_ex_price as u64;
    self.vault_borrow_exchange_price = vault_borrow_ex_price as u64;
    [...]
}
```

The administrative magnifier setters checkpoint accrued exchange prices before storing the replacement magnifier. As shown by `update_supply_rate_magnifier`, the setter calls `user::update_exchange_prices` before `set_supply_rate_magnifier`. The borrow-rate magnifier setter follows the same ordering.

packages/vaults/sources/module/admin.move

```
public fun update_supply_rate_magnifier<Col, Debt>([...]) {
    auth::assert_auths(vault_registry, vault_admin, ctx);

    // Update exchange price first
```

© 2026 Otter Audits LLC. All Rights Reserved.

```
    user::update_exchange_prices(vault, vault_registry, col_reserve, debt_reserve,
    clock);

    vault.set_supply_rate_magnifier(supply_rate_magnifier);

    events::emit_log_update_supply_rate_magnifier(
        supply_rate_magnifier.magnifier_magnitude(),
        supply_rate_magnifier.magnifier_is_negative(),
    );
}
```

Once an accrued exchange price no longer fits in `u64`, the mandatory checkpoint aborts before the new magnifier is stored. An administrator therefore cannot replace a problematic supply or borrow rate magnifier through these setters, potentially leaving the vault unable to correct the configuration responsible for continued accrual.

Remediation

Store all four checkpointed exchange prices as `u128` values in `VaultMarket` and remove the narrowing casts from initialization and `update_exchange_prices`. Update the corresponding getters and migration logic for existing vault objects so both calculated and persisted prices use the same width, allowing administrative setters to checkpoint the old configuration before installing a replacement.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Premature u64 Narrowing Blocks Capped Rebalances

ID	Severity	Status
OS-FLV-ADV-01	Medium	Ack'd

Description

The vault calculates each collateral and debt imbalance from two `u128` totals, but `compute_rebalance_amount` converts the complete difference to the Liquidity layer's `u64` `Amount` representation immediately. When the difference exceeds the maximum `u64` value, this narrowing conversion aborts before the rebalance logic can select a smaller amount.

packages/vaults/sources/module/admin.move

```
533 fun compute_rebalance_amount(total_vault: u128, total_external: u128): Amount {
534     if (total_vault > total_external) {
535         liq_structs::positive(((total_vault - total_external) as u64))
536     } else if (total_external > total_vault) {
537         liq_structs::negative(((total_external - total_vault) as u64))
538     } else {
539         liq_structs::none()
540     }
541 }
```

Although `rebalance_with_amounts` accepts optional `u128` caps for both legs, it invokes `get_adjusted_rebalance_amount` only after obtaining the full `Amount`. As shown in `get_adjusted_rebalance_amount`, the function therefore receives a magnitude that has already been narrowed to `u64`. Its later conversion back to `u128` cannot prevent the earlier abort.

packages/vaults/sources/module/admin.move

```
1  fun get_adjusted_rebalance_amount(full: Amount, custom: Option<u128>): Amount {
2      if (custom.is_none()) {
3          return full
4      };
5      let custom = *custom.borrow();
6      let magnitude = full.value() as u128;
7      let capped = (if (magnitude < custom) magnitude else custom) as u64;
8
9      if (full.get_is_positive()) {
10         liq_structs::positive(capped)
11     } else if (full.get_is_negative()) {
12         liq_structs::negative(capped)
13     } else {
14         liq_structs::none()
15     }
16 }
```

© 2026 Otter Audits LLC. All Rights Reserved.

Consequently, an authorized rebalancer cannot execute an otherwise representable partial settlement when the full collateral or debt gap exceeds the `u64` range, even if the supplied custom cap fits within that range. This can leave the affected vault-to-liquidity imbalance unrebalanced through the bounded entry point until the underlying gap is reduced.

Remediation

Preserve the imbalance direction and absolute magnitude in `u128` until after applying the optional caller-provided cap. Convert only the selected settlement magnitude to `u64`, using an explicit checked range assertion before constructing the Liquidity `Amount`. Define the uncapped behavior for gaps above the `u64` range explicitly, such as selecting a `u64`-sized partial settlement or returning a dedicated range error, while ensuring that a representable custom cap can always be applied before conversion.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Maximum Withdrawal Desynchronizes Collateral Position Accounting

ID	Severity	Status
OS-FLV-ADV-02	Low	Ack'd

Description

`position::get_new_position_info` does not resolve a maximum collateral withdrawal to the full collateral value. It sets the position's raw collateral to zero, but converts calculated withdrawal values of zero or one to `ScaledAmount::None` and reduces every larger value by one. As a result, the settlement instruction does not consistently match the zero collateral balance recorded for the position.

packages/vaults/sources/state/position.move

```
public(package) fun get_new_position_info(
    mut info: AmountInfo,
    exchange_prices: VaultExchangePrices,
): (ScaledAmount, ScaledAmount, u128, u128) {
    [...]
    } else if (info.new_col.get_scaled_is_max()) {
        // max withdraw, rounding up:
        // adding +1 to negative withdrawAmount newCol_ for safe rounding (reducing withdraw)
        let withdraw_amount: u128 = math::mul_div_u128(
            info.col_raw,
            exchange_prices.supply_ex_price,
            epp,
        );

        info.col_raw = 0;

        // `withdraw_amount` of 0 or 1 both collapse to `None` here: tagging either as a real
        // `Positive` would route it down the supply side as a deposit instead of a no-op.
        if (withdraw_amount <= 1) {
            structs::scaled_none()
        } else {
            structs::scaled_negative(withdraw_amount - 1)
        }
    }
    [...]
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

`operate` rejects the operation when both resolved sides are `None`. Consequently, a collateral-only maximum withdrawal whose calculated value is zero or one aborts at this guard instead of reaching the position update and settlement logic.

packages/vaults/sources/module/user.move

```
public fun operate<Col, Debt>([...]): (Balance<Col>, Balance<Debt>,
Option<PositionOwnerCap>) {
    [...]
    assert!(
        !(new_col_final.scaled_is_none() && new_debt_final.scaled_is_none()),
        errors::vault_invalid_operate_amount(),
    );
    [...]
}
```

Larger dust withdrawals can fail later when the reduced collateral amount is sent to Liquidity. The collateral settlement call has no borrow leg, and `operate_internal` aborts when both operation values are below `min_operate_amount`, which is ten native token units. Thus, whenever the unscaled maximum-withdrawal amount remains below that threshold after the subtraction, the position cannot be cleared through this path.

packages/liquidity/sources/module/user.move

```
fun operate_internal<T>([...]): (Balance<T>, Balance<T>, u64, u64) {
    [...]
    if (
        supply_amount.value() < constants::min_operate_amount()
            && borrow_amount.value() < constants::min_operate_amount()
    ) {
        abort errors::operate_amounts_nearly_zero()
    };
    [...]
}
```

The deletion gate in `position::delete` requires a supply-only position with a zero recorded supply. Failed maximum withdrawals do not reach the state update needed to satisfy this condition, preventing affected dust positions from being closed. Conversely, when the reduced withdrawal succeeds, the position is recorded with zero collateral even though the amount settled through Liquidity was reduced, leaving collateral residue outside the position's recorded balance. Users may therefore be unable to recover or cleanly account for the full collateral represented before the maximum withdrawal.

© 2026 Otter Audits LLC. All Rights Reserved.

packages/vaults/sources/state/position.move

```
public(package) fun delete(self: Position, cap: PositionOwnerCap): (ID, ID) {
    [...]
    assert!(
        self.is_supply_only_position && self.supply_amount == 0,
        errors::vault_position_not_empty(),
    );
    [...]
}
```

Remediation

Resolve a maximum collateral withdrawal from the amount that can actually be settled, and update `col_raw` only by the corresponding settled raw amount. Do not unconditionally set `col_raw` to zero while returning `None` or an amount reduced by one. Where the full floor-converted withdrawal is safe, pass that full amount to Liquidity; if conservative rounding must retain a remainder, preserve that remainder in the position instead of permitting deletion. Add an explicit dust-clearing mechanism for nonzero balances whose native-token value is below Liquidity's minimum operation amount, and permit deletion only after that dust has been withdrawn or deliberately reconciled in vault accounting. Cover calculated withdrawal values around zero, one, `min_operate_amount - 1`, `min_operate_amount`, and `min_operate_amount + 1` in regression tests.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Missing Vault Asset Compatibility Validation

ID	Severity	Status
OS-FLV-ADV-03	Low	Ack'd

Description

The vault initialization path does not fully validate that its asset configuration is compatible with Liquidity. In `vault_market::create`, both supplied decimal values are checked only against the maximum supported value. Consequently, zero-decimal and one-decimal values pass vault creation.

packages/vaults/sources/state/vault_market.move

```
public(package) fun create<Col, Debt>(
    registry: &mut VaultRegistry,
    risk: RiskConfig,
    supply_rate_magnifier: RateMagnifier,
    borrow_rate_magnifier: RateMagnifier,
    oracle: ID,
    liquidity_program: ID,
    col_decimals: u8,
    debt_decimals: u8,
    clock: &Clock,
    ctx: &mut TxContext,
): VaultMarket<Col, Debt> {
    assert!(
        col_decimals <= constants::max_token_decimals()
            && debt_decimals <= constants::max_token_decimals(),
        errors::vault_invalid_decimals(),
    );
    [...]
}
```

Liquidity defines two decimals as the inclusive minimum for listed tokens, and `reserve::new_reserve` enforces this lower bound when creating a token reserve. The missing equivalent lower-bound check in the vault permits vault configuration outside the decimal range accepted during Liquidity reserve creation.

packages/liquidity/sources/constants.move

```
/// Minimum token decimals for any token that can be listed at Liquidity (inclusive).
public(package) fun min_token_decimals(): u8 { 2 }
```

© 2026 Otter Audits LLC. All Rights Reserved.

Vault creation also does not require `Col` and `Debt` to be different types. The subsequent `vault_market::add_caps` flow accepts a collateral `UserCap<Col>` and a debt `UserCap<Debt>`, writes them to distinct collateral and debt dynamic-field slots, and considers capability installation complete only when both slots are populated.

packages/vaults/sources/state/vault_market.move

```
public(package) fun add_caps<Col, Debt>(
    self: &mut VaultMarket<Col, Debt>,
    col_cap: Option<UserCap<Col>>,
    debt_cap: Option<UserCap<Debt>>,
) {[...]}
```

Liquidity defines `UserCap<T>` as a derived object and mints it by claiming the key produced by `keys::user_cap_key<T>(holder)`. Capability identity is therefore determined by the holder and token type. If `Col` and `Debt` are identical, the market cannot obtain two independent capability objects for the same holder and type to populate both permanent slots.

packages/liquidity/sources/state/user_cap.move

```
/// Possession-based identity of a calling protocol account.
public struct UserCap<phantom T> has key, store {
    /// Derived: claimed under the `Registry` at `keys::user_cap_key<T>(holder)`. The claim
    /// aborts on re-claim, so at most one cap per `(holder, T)` can ever exist.
    id: UID,
    /// The account this cap names. Written once at mint.
    holder: Holder,
}
```

These omissions allow an authorized configuration transaction to create a vault that is inconsistent with Liquidity's constraints. An equal-asset market cannot complete capability installation and therefore cannot become operational.

Remediation

Require both `col_decimals` and `debt_decimals` to be at least Liquidity's minimum supported token decimals as well as no greater than the existing maximum in `vault_market::create`. Before allocating and sharing the market, also reject configurations for which the defining type identities of `Col` and `Debt` are equal. Where possible, source or verify each supplied decimal value against the corresponding Liquidity listing rather than accepting an independent governance-provided value.

Patch

This issue was fixed in PR#55, and `vault_market::create` now enforces `2 <= decimals <= 9` for both coins and prevents registration of an incompatible market by requiring `Col` and `Debt` to be

© 2026 Otter Audits LLC. All Rights Reserved.

distinct. However, decimals remain admin-supplied `u8` parameters rather than values read from `CoinMetadata`, leaving incorrect in-range values as a trusted-admin misconfiguration, which Fluid acknowledged.

© 2026 Otter Audits LLC. All Rights Reserved.

Collateral-Factor Check Rejects Net Deleveraging Operations

ID	Severity	Status
OS-FLV-ADV-04	Low	✓ Resolved

Description

The `operate` flow applies both requested amounts, derives the resulting position tick, and then calls `check_if_position_safe`. As shown below, `check_if_position_safe` selects the stricter collateral-factor boundary whenever collateral is withdrawn, or debt is borrowed. This classification considers the direction of either operation independently, but does not compare the position's previous leverage with its resulting leverage.

packages/vaults/sources/utils/operate.move

```
public(package) fun check_if_position_safe<Col, Debt>(
    vault: &VaultMarket<Col, Debt>,
    quote: &PriceQuote,
    tick: TickIndex,
    supply_ex_price: u128,
    borrow_ex_price: u128,
    new_col: &ScaledAmount,
    new_debt: &ScaledAmount,
) {
    [...]
    let (exchange]_rate, threshold, error_code) = if (
        new_col.get_scaled_is_negative() || new_debt.get_scaled_is_positive()
    ) {
        // withdraw or borrow, check collateral factor
        (
            exchange_rate_operate,
            vault.collateral_factor() as u128,
            errors::vault_position_above_cf(),
        )
    } else {
        // deposit or payback, check liquidation threshold
        (
            exchange_rate_liquidate,
            vault.liquidation_threshold() as u128,
            errors::vault_position_above_liquidation_threshold(),
        )
    };
    [...]
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

The function then selects a limit tick from the chosen exchange rate and threshold and asserts that the resulting tick does not exceed that limit. Although the comments expressly discuss allowing operations that reduce the ratio through deleveraging even when collateral also decreases, the function performs no old-to-new ratio comparison. Consequently, a combined withdrawal and larger repayment can reduce the position's leverage yet still be rejected if its resulting tick remains above the collateral-factor boundary. This prevents an otherwise safety-improving combined operation and may obstruct users attempting to deleverage a position through a single `operate` call.

Remediation

Classify mixed operations according to their net effect on position safety rather than solely from the presence of a withdrawal or borrow. Pass the pre-operation position ratio or tick into the health check and compare it with the resulting ratio or tick. Continue enforcing the collateral-factor boundary for operations that increase leverage; for an operation that strictly decreases leverage, apply the intended deleveraging rule and its appropriate safety boundary.

Patch

Resolved in PR#52.

© 2026 Otter Audits LLC. All Rights Reserved.

Liquidation-Only Quotes Rejected During Position Improvements

ID	Severity	Status
OS-FLV-ADV-05	Low	✓ Resolved

Description

The position-safety check obtains its oracle rates through `get_both_exchange_rate`. It unconditionally reads both `quote.rate_liquidate()` and `quote.rate_operate()`.

packages/vaults/sources/invokes/oracle.move

```
public(package) fun get_both_exchange_rate<Col, Debt>(
    vault: &VaultMarket<Col, Debt>,
    quote: &PriceQuote,
): (u128, u128) {
    assert_market_oracle(vault, quote);
    (quote.rate_liquidate(), quote.rate_operate())
}
```

This occurs before `check_if_position_safe` selects the applicable rate. Withdrawals and borrows use the operate rate and collateral factor, whereas collateral deposits and partial repayments use only the liquidation rate and liquidation threshold. Consequently, the latter paths request an operate rate that their safety calculation does not consume. The oracle package deliberately permits liquidation-only quotes whose operate rate is absent, and accessing that absent rate aborts. Supplying such a quote to a collateral deposit or partial repayment that leaves debt outstanding therefore aborts the operation, even though the available liquidation rate is sufficient for the intended safety check.

Remediation

Select the applicable safety-check branch before retrieving an exchange rate.

Patch

Resolved in PR#52.

© 2026 Otter Audits LLC. All Rights Reserved.

Vault ID Missing from Configuration Update Events

ID	Severity	Status
OS-FLV-ADV-06	Low	✓ Resolved

Description

In the current implementation, `LogUserPosition` event includes a `vault_id` alongside its position and accounting data, allowing consumers to attribute the event to a particular vault.

packages/vaults/sources/types/events.move

```
public struct LogUserPosition has copy, drop {
    user: address,
    nft_id: ID,
    vault_id: u64,
    position_mint: ID,
    tick: Tick,
    col: u64,
    borrow: u64,
}
```

However, the events for supply and borrow rate magnifiers, collateral and liquidation risk parameters, withdrawal gap, borrow fee, core settings, oracle changes, and exchange-price updates contain the changed values without a vault identifier. Their emitters are called while modifying a specific `VaultMarket`, but that identity is not included in the emitted payload. Consequently, when multiple vaults emit the same event type, off-chain indexers and monitoring systems cannot attribute an update from the event payload alone and must reconstruct the affected vault from external transaction context. This may lead to ambiguous or incorrectly associated configuration histories.

Remediation

Add a `vault_id` field to each vault-specific risk, oracle, magnifier, core-settings, and exchange-price update event. Update the corresponding emitter functions to accept this identifier, and pass `vault.vault_id()` from every administrative setter and from `VaultMarket::update_exchange_prices`.

Patch

Resolved in PR#55.

© 2026 Otter Audits LLC. All Rights Reserved.

Inconsistent Rounding Weakens the Withdrawal Gap

ID	Severity	Status
OS-FLV-ADV-07	Low	Ack'd

Description

The vault's `check_if_withdrawal_safe_for_withdrawal_gap` converts the requested token withdrawal into raw liquidity units using floor division before comparing it with the headroom between the vault's adjusted supply and its liquidity withdrawal limit. However, the liquidity layer's `supply_or_withdraw` applies ceiling division when converting the same withdrawal for debiting the vault's supply position. When the conversion has a nonzero remainder, the preliminary check therefore assesses one fewer raw unit than liquidity subsequently removes.

The vault performs this withdrawal-gap check before calling the liquidity settlement path. A withdrawal at the calculated headroom boundary may therefore pass the vault check while the liquidity layer debits one additional raw unit, leaving the position one raw unit below the buffer intended by the withdrawal-gap calculation. Although the discrepancy is limited to integer-rounding residue, it weakens the configured protection at boundary values.

Remediation

Convert `withdraw_token_amount` to raw liquidity units with ceiling division in `check_if_withdrawal_safe_for_withdrawal_gap`, matching the conversion used by the liquidity layer for withdrawals.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Inconsistent Collateral Rounding Across Accounting Layers

ID	Severity	Status
OS-FLV-ADV-08	Low	Ack'd

Description

Vault markets support tokens with fewer than nine decimals by multiplying native token amounts by a decimal scale before updating their internal collateral accounting. For a collateral deposit, `position::get_new_position_info` then converts this scaled amount into raw vault collateral using the vault supply exchange price and floor rounding.

The amount subsequently forwarded through `fluid_liquidity::user::operate` depends on the deposit path. When the deposit uses the full input balance, `unscale_col` returns the unscaled native amount. When the input has a spare balance, `unscale_col` instead returns the unscaled amount plus one native unit. The liquidity supply position then independently converts the forwarded native amount into raw liquidity shares using its supply exchange price and floor rounding.

The independently rounded vault and liquidity calculations can credit quantities that do not correspond exactly. These rounding residues can cause the vault's recorded collateral and its liquidity-layer supply position to diverge across deposits.

Remediation

Use a canonical accounting relationship that explicitly handles both `unscale_col` branches.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Placement Dust Inflates Absorbed Debt

ID	Severity	Status
OS-FLV-ADV-09	Low	Ack'd

Description

The vault uses inconsistent debt values for tick accounting and aggregate borrow accounting. As shown in `compute_placement`, placing a position derives a tick-level `user_raw_debt` above the requested net debt and records the difference as `dust_debt`. The resulting gross debt is added to the tick ladder, while the dust remains associated with the position.

packages/vaults/sources/utils/operate.move

```
public fun compute_placement(net_debt_raw: u128, total_col_raw: u128): (TickIndex, u128, u128) {
    // Calculate ratio
    let ratio: u128 = math::mul_div_u128(
        net_debt_raw,
        tick_math::zero_tick_scaled_ratio(),
        total_col_raw,
    );
    // Get tick at ratio (floor)
    let (tick_at_ratio, ratio_at_tick) = tick_math::get_tick_at_ratio(ratio);
    // Increase tick by 1 and ratio by one tick (1.0015x)
    let tick = tick_at_ratio.add_u32(1);
    assert!(tick.is_in_bounds(), errors::vault_tick_has_debt_out_of_range());
    let ratio_new: u128 = math::mul_div_u128([...]);
    // Calculate user's debt with the adjusted ratio
    let user_raw_debt: u128 = (ratio_new * total_col_raw) >> tick_math::shift();
    let dust_debt: u128 = user_raw_debt - net_debt_raw;
    (tick, user_raw_debt, dust_debt)
}
```

When the position is subsequently loaded, `get_position_info` reconstructs its raw debt from the tick ratio and collateral, applying upward rounding to both the collateral and calculated debt as shown below. The stored dust is returned separately, allowing the operate path to derive the position's net debt by subtracting that dust from the reconstructed amount.

packages/vaults/sources/state/position.move

```
public fun get_position_info(self: &Position): (u128, u128, u128, TickIndex, u32) {
    [...]
    let debt_raw: u128 = if (tick.gt(tick_math::cold_tick())) {
        // Round up the collateral for debt calculation
        let collateral_for_debt_calc: u128 = col_raw + 1;

```

© 2026 Otter Audits LLC. All Rights Reserved.

```
        // Fetch current debt based on tick ratio and collateral
        // rounding up the debt
        ((tick_math::get_ratio_at_tick(tick) * collateral_for_debt_calc)
            >> tick_math::shift()) + 1
    }
    [...]
}
```

The aggregate update in `operate` passes `get_net_debt_raw()` to `update_total_borrow`. Because `get_net_debt_raw()` subtracts the position's stored dust, `total_borrow` represents net position debt even though the corresponding tick stores the gross placement debt.

packages/vaults/sources/module/user.move

```
public fun operate<Col, Debt>([...]): (Balance<Col>, Balance<Debt>,
Option<PositionOwnerCap>) {
    [...]
    // --- totals ---
    vault.update_total_supply(memory_vars.col_raw(), old_state.old_col_raw());
    vault.update_total_borrow(memory_vars.get_net_debt_raw(),
    old_state.old_net_debt_raw());
    [...]
}
```

This distinction is lost when an insolvent tick is absorbed. As shown in `fetch_next_tick_absorb`, the function reads the tick's complete raw debt into `tick_debt` and adds it directly to `debt_absorbed`. The tick does not expose the portion attributable to stored position dust at this point, so the accumulated absorbed debt includes placement dust that was never included in `total_borrow`.

packages/vaults/sources/state/tick.move

```
public(package) fun fetch_next_tick_absorb(
    bitmap: &mut Table<u16, u256>,
    ticks: &mut Table<TickIndex, Tick>,
    current_tick: TickIndex,
    max_tick: TickIndex,
): (TickIndex, u128, u128) {
    [...]
    // For last user remaining in vault there could be a lot of loop iterations
    // Chances of this to happen is extremely low (like ~0%)
    loop {
        [...]
        let tick_debt = {
            let tick_data = load_tick_mut(ticks, next_tick);
            let tick_debt = tick_data.get_raw_debt();
```

© 2026 Otter Audits LLC. All Rights Reserved.

```
            tick_data.set_fully_liquidated();
            tick_debt
        };

        let ratio = tick_math::get_ratio_at_tick(next_tick);

        debt_absorbed = debt_absorbed + tick_debt;
        [...]
    }
}
```

The `absorb` path then calls `add_absorbed_debt_amount(debt_absorbed)`. A later liquidation can use that absorbed-debt balance as settled debt and pass it to `reduce_total_borrow`. If the dust-inclusive amount exceeds the net borrow recorded by the vault, the checked subtraction may underflow and abort. Consequently, placement dust can prevent affected absorbed debt from being settled.

packages/vaults/sources/module/user.move

```
fun absorb<Col, Debt>(vault: &mut VaultMarket<Col, Debt>, max_tick: TickIndex): (u128, u128) {
    [...]
    vault.add_absorbed_col_amount(col_absorbed);
    vault.add_absorbed_debt_amount(debt_absorbed);
    [...]
}
```

Remediation

Use the same net-debt basis for tick absorption and `total_borrow`. If ticks must retain gross debt for ratio and liquidation calculations, add per-tick dust accounting: increment it when a position's gross debt is placed, decrement it when that position leaves the tick, and clear it together with the tick during absorption. Have `fetch_next_tick_absorb` return only `tick_raw_debt - tick_dust` as absorbable debt, and add only that net value to `absorbed_debt_amount`.

Patch

Fluid acknowledged this issue.

© 2026 Otter Audits LLC. All Rights Reserved.

Narrow LogAbsorb Fields Abort Large Absorptions

ID	Severity	Status
OS-FLV-ADV-10	Low	✓ Resolved

Description

The absorption logic aggregates collateral and debt across eligible ticks using `u128` values. As shown in `fetch_next_tick_absorb`, each absorbed tick increases the running `col_absorbed` and `debt_absorbed` totals, and the function returns both totals without restricting them to the `u64` range.

packages/vaults/sources/state/tick.move

```
public(package) fun fetch_next_tick_absorb(
    bitmap: &mut Table<u16, u256>,
    ticks: &mut Table<TickIndex, Tick>,
    current_tick: TickIndex,
    max_tick: TickIndex,
): (TickIndex, u128, u128) {
    [...]
    // For last user remaining in vault there could be a lot of loop iterations
    // Chances of this to happen is extremely low (like ~0%)
    loop {
        [...]
        let tick_debt = {
            let tick_data = load_tick_mut(ticks, next_tick);
            let tick_debt = tick_data.get_raw_debt();
            tick_data.set_fully_liquidated();
            tick_debt
        };
        [...]
    }
}
```

When `liquidate` detects that the vault's topmost tick exceeds `max_tick`, it calls `absorb` and then unscales the returned totals for `emit_log_absorb`. The event accepts `u64` fields, so both unscaled `u128` values are narrowed with `as u64`. If either value exceeds the `u64` range, this conversion aborts.

packages/vaults/sources/module/user.move

```
public fun liquidate<Col, Debt>([...]): (Balance<Col>, Balance<Debt>) {
    [...]
    if (vault.topmost_tick().gt(max_tick)) {
        let (col_absorbed, debt_absorbed) = absorb(vault, max_tick);

```

© 2026 Otter Audits LLC. All Rights Reserved.

```
        events::emit_log_absorb(
            constants::unscale_amounts(col_absorbed, supply_decimals) as u64,
            constants::unscale_amounts(debt_absorbed, debt_decimals) as u64,
        );

        if (debt_amt_scaled == 0) {
            debt_in.destroy_zero();
            return (balance::zero<Col>(), balance::zero<Debt>())
        };
    };
    [...]
}
```

Because the event is emitted immediately after absorption and before the remaining liquidation logic or the absorb-only early return, an out-of-range event value blocks the entire liquidation call. This may prevent a caller from processing positions above `max_tick` even though the underlying absorption accounting supports the larger totals.

Remediation

Change the `LogAbsorb` collateral and debt fields and `emit_log_absorb` parameters to `u128`, then pass the unscaled absorption totals without narrowing casts. Add boundary coverage for unscaled totals at and above the `u64` maximum to confirm that event emission cannot abort an otherwise valid absorption.

Patch

Resolved in PR#55.

© 2026 Otter Audits LLC. All Rights Reserved.

5. Suggestions

Here, we present a discussion of the 2 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-FLV-SUG-00	Informational	✓ Resolved	Low Oracle Prices Reach Unsupported Tick Math
OS-FLV-SUG-01	Informational	○ Ack'd	Lack of Functionality to Borrow Stored Display Capability

© 2026 Otter Audits LLC. All Rights Reserved.

Low Oracle Prices Reach Unsupported Tick Math

ID	Severity	Status
OS-FLV-SUG-00	Informational	✓ Resolved

Description

Currently, the oracle implementation does not version-gate its functions. This is particularly impactful for view functions that compute prices and return a `PriceQuote`, as they remain callable regardless of the active oracle version. Functions such as `get_both_exchange_rate`, `get_exchange_rate_operate`, and `get_exchange_rate_liquidate` accept only `&Oracle`, `&PriceAdapter`, `&PriceInfoObject`, and `&Clock`, and neither `Oracle` nor `PriceQuote` carries version information.

packages/vaults/sources/types/events.move

```
public struct LogUserPosition has copy, drop {
    user: address,
    nft_id: ID,
    vault_id: u64,
    position_mint: ID,
    tick: Tick,
    col: u64,
    borrow: u64,
}
```

As a result, no read path invokes `registry.assert_version()`. Because Sui upgrades preserve type identity while keeping earlier package versions callable, users can invoke outdated view functions even after their pricing logic has been fixed. Those legacy functions can produce a `PriceQuote` indistinguishable from one generated by the latest implementation, allowing security fixes introduced by an upgrade to be bypassed.

Remediation

Require every quote function to call `registry.assert_version()`, and bind the validated version to `PriceQuote` so downstream vaults can verify that the quote was produced by the expected oracle version.

Patch

Resolved in PR#57.

© 2026 Otter Audits LLC. All Rights Reserved.

Lack of Functionality to Borrow Stored Display Capability

ID	Severity	Status
OS-FLV-SUG-01	Informational	Ack'd

Description

The protocol does not expose a way to borrow the stored display capability after initialization, making future metadata or presentation updates impossible. If names, descriptions, images, or other display fields must be corrected, administrators cannot access the capability required to authorize those edits.

Remediation

Add functionality to mutably borrow the stored display capability for future metadata updates.

Patch

Fluid acknowledged this issue.

© 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.