OtterSec

Fluid Lending Rewards

Security Assessment

Prepared for
Fluid

Prepared by
Otter Audits, LLC

Prepared on
September 11th, 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-FLR-ADV-00 Withdrawals May Burn Insufficient Lending Shares 6
5. Suggestions 7
OS-FLR-SUG-00 Rounded-Down TVL Inflates Reward Accrual 8
OS-FLR-SUG-01 Code Clarity 9
OS-FLR-SUG-02 Inconcistencies in Validation Logic 10
OS-FLR-SUG-03 Code Maturity 11
A. Vulnerability Rating Scale 13
B. Procedure 14

© 2026 Otter Audits LLC. All Rights Reserved.

1. Executive Summary

1.1. Overview

Fluid engaged OtterSec to conduct a security assessment of the `lending-protocol` and `lending-rewards` programs. This assessment was conducted between August 24th and September 4th, 2026. For information on our auditing methodology, refer to Appendix B.

1.2. Key Findings

We produced 5 findings throughout this audit engagement.

In particular we identified that withdrawals burn shares based on the requested amount rather than the actual liquidity layer position reduction, allowing rounding differences to under-burn shares. The resulting accounting shortfall must be covered by the protocol during rebalancing (● OS-FLR-ADV-00). Additionally, rounding TVL down understates the reward-rate divisor, inflating rewards and the fToken exchange price (● OS-FLR-SUG-00).

We also made suggestions to improve the overall code clarity (● OS-FLR-SUG-01), and address inconcistencies in the validation logic (● OS-FLR-SUG-02). Lastly, we reccomended refactoring the code to remove redundant code logic (● OS-FLR-SUG-03).

© 2026 Otter Audits LLC. All Rights Reserved.

2. Scope

The `lending-protocol` is the core lending protocol that creates fTokens (yield-bearing tokens) for each supported asset. Users deposit underlying tokens to receive fTokens and earn yields, while the protocol tracks exchange rates, manages deposits, and routes funds through integration with the liquidity layer.

The `lending-rewards` implementation rewards distribution system that manages incentive campaigns for lending vaults. Allows admins to create, schedule, and manage reward distributions to lenders, with features to start campaigns, queue future campaigns, and calculate reward rates based on TVL and duration.

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

This audit was performed against PR#8 and PR#9.

© 2026 Otter Audits LLC. All Rights Reserved.

3. Findings

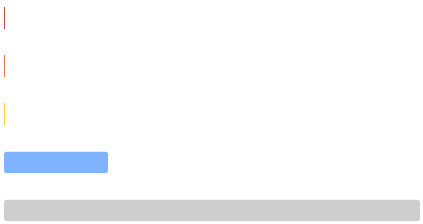
Findings were rated using the criteria in Appendix A.

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

3.1. Findings Summary

Severity	Count
Critical	0
High	0
Medium	0
Low	1
Informational	4



© 2026 Otter Audits LLC. All Rights Reserved.

4. Advisories

Here, we present a technical analysis of the 1 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-FLR-ADV-00	Low	✓ Resolved	Withdrawals May Burn Insufficient Lending Shares

© 2026 Otter Audits LLC. All Rights Reserved.

Withdrawals May Burn Insufficient Lending Shares

ID	Severity	Status
OS-FLR-ADV-00	Low	✓ Resolved

Description

The withdrawal flow calculates and burns the caller's lending shares from the requested asset amount rather than from the supply reduction actually registered by the Fluid Liquidity layer. Because exchange-price conversions use integer rounding, the liquidity layer may deduct a greater underlying value than the shares burned represent. This discrepancy leaves the lending protocol with fewer reserve assets without a corresponding reduction in its share liabilities, requiring the protocol to absorb the shortfall during rebalancing.

Remediation

Ensure the withdrawal mirrors the deposit approach by comparing the raw liquidity position before and after the operation and using that actual delta to determine the precise number of shares to burn.

Patch

Resolved in 00cddb4.

© 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-FLR-SUG-00	Informational	✓ Resolved	Rounded-Down TVL Inflates Reward Accrual
OS-FLR-SUG-01	Informational	✓ Resolved	Code Clarity
OS-FLR-SUG-02	Informational	✓ Resolved	Inconcistencies in Validation Logic
OS-FLR-SUG-03	Informational	✓ Resolved	Code Maturity

© 2026 Otter Audits LLC. All Rights Reserved.

Rounded-Down TVL Inflates Reward Accrual

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

Description

`liquidity_calcs::accrued_token_exchange_price` calculates TVL with `to_assets(..., false)`, which rounds the `fToken` liability down. Because TVL is used as the divisor when deriving the reward rate, this understated value inflates the rate and causes excessive rewards to accrue into the token exchange price. TVL should be rounded up to conservatively prevent `fToken` holders from being credited more reward yield than intended.

Remediation

Round-up the TVL to prevent over-crediting holders.

Patch

Resolved in 7664732.

© 2026 Otter Audits LLC. All Rights Reserved.

Code Clarity

ID	Severity	Status
OS-FLR-SUG-01	Informational	Resolved

Description

1. The comment for the `max_rewards_rate` constant incorrectly states that `50_000_000_000_000` represents `50% * exchange_prices_precision`. However, the value is actually scaled using `return_percent_precision`.

packages/lending/sources/constants.move

```
12 public(package) fun max_rewards_rate(): u64 { 50_000_000_000_000 } // 50% *
   exchange_prices_precision
```

2. `user::update_rate` returns `(token_exchange_price, liquidity_exchange_price)`, whereas `lending_state::exchange_prices` returns `(liquidity_exchange_price, token_exchange_price)`. Because both values share the same `u64` type, callers can silently interpret them in the wrong order without a type error. Standardize the tuple ordering across both functions.

3. `accounting::preview_mint` returns only the assets required for the requested shares, although Fluid Liquidity's asset-to-raw rounding may cause the actual minted share amount to differ. Return the expected shares alongside the required assets to let callers set accurate slippage bounds and reliably predict execution results.

packages/lending/sources/helpers/accounting.move

```
185 public fun preview_mint<T>(
186     state: &LendingState<T>,
187     reserve: &TokenReserve<T>,
188     shares: u64,
189     clock: &Clock,
190 ): u64 {
191     let token_exchange_price = calculate_new_token_exchange_price(state,
    reserve, clock);
192     lending_calcs::to_assets(shares, token_exchange_price, true)
193 }
```

Remediation

Incorporate the suggestions to improve the overall code clarity.

Patch

1. Issue #1 resolved in 692e5d7.
2. Issue #2 resolved in 692e5d7.
3. Issue #3 resolved in 692e5d7.

© 2026 Otter Audits LLC. All Rights Reserved.

Inconcistencies in Validation Logic

ID	Severity	Status
OS-FLR-SUG-02	Informational	✓ Resolved

Description

1. `create_manager` in `lending_rewards::admin` creates and shares a vault-bound `RewardsManager<T>` without emitting an event, reducing observability for indexers and administrators. It also checks only that the capability references the correct vault, allowing a stale `RewardsWriterCap<T>` for that vault to be installed. The function should verify that `object::id(&cap)` matches `lending_state::rewards_writer()` and emit a manager-creation event.

   packages/lending_rewards/sources/module/admin.move

```
47 public fun create_manager<T>(
48     rewards_auth: &RewardsAuth,
49     vault: &LendingState<T>,
50     cap: RewardsWriterCap<T>,
51     ctx: &mut TxContext,
52 ) {
53     rewards_auth.assert_version();
54     rewards_auth.assert_auth(ctx);
55     let state_id = object::id(vault);
56     assert!(cap.cap_state_id() == state_id, errors::invalid_params());
57     rewards_manager::new(cap, state_id, ctx).share();
58 }
```

2. `rebalance` in `fluid_lending::admin` still calls `liquidity::supply` when the calculated supplied amount is zero, such as when no backing gap exists or the provided balance is empty. The Liquidity Layer rejects zero-value supply operations, causing the transaction to fail indirectly with an external error. The function should instead short-circuit and abort before the external call.

Remediation

Update the validation logic as stated above.

Patch

1. Issue #1 resolved in 00cddb4.
2. Issue #2 resolved in 692e5d7.

© 2026 Otter Audits LLC. All Rights Reserved.

Code Maturity

ID	Severity	Status
OS-FLR-SUG-03	Informational	Resolved

Description

1. `lending_rewards::admin::create_manager` accepts `vault` only to obtain its object ID and verify it against `cap.cap_state_id()`. Because the capability already carries the authoritative lending-state ID, the function could derive `state_id` directly from the cap and remove the redundant vault argument.

packages/lending_rewards/sources/module/admin.move

```
47  public fun create_manager<T>(
48      rewards_auth: &RewardsAuth,
49      vault: &LendingState<T>,
50      cap: RewardsWriterCap<T>,
51      ctx: &mut TxContext,
52  ) {
53      rewards_auth.assert_version();
54      rewards_auth.assert_auth(ctx);
55      let state_id = object::id(vault);
56      assert!(cap.cap_state_id() == state_id, errors::invalid_params());
57      rewards_manager::new(cap, state_id, ctx).share();
58  }
```

2. In `fluid_lending::rewards`, `get_rate_segments` calls `rate_for` twice even though both campaigns use the same `total_assets` and `start_tvl` eligibility check. Instead it should immediately return an empty vector when `total_assets == 0` or `total_assets < self.start_tvl`, avoiding redundant rate calculations and segment-bound processing.

packages/lending/sources/state/rewards.move

```
98  public fun get_rate_segments(
99      self: &RewardsSchedule,
100     total_assets: u64,
101     from_s: TimestampS,
102     now_s: TimestampS,
103 ): vector<RateSegment> {
104     let mut segments = vector[];
105     push_segment(
106         &mut segments,
107         rate_for(self.yearly_reward, total_assets, self.start_tvl),
108         from_s.max_ts(self.start_s),
109         now_s.min_ts(self.end_s),
110     );
111     push_segment(
```

© 2026 Otter Audits LLC. All Rights Reserved.

```
            &mut segments,
            rate_for(self.next_yearly_reward, total_assets, self.start_tvl),
            from_s.max_ts(self.next_start_s),
            now_s.min_ts(self.next_end_s),
        );
        segments
    }
```

3. `rewards_calcs::annualize` currently calls `mul_div_u128` and then manually verifies that the result fits within `u64` before casting it. Instead, use `mul_div_u64` directly to preserve the required checked arithmetic while removing the explicit bounds check and conversion, simplifying the implementation.

packages/lending_rewards/sources/math/rewards_calcs.move

```
public fun annualize(reward_amount: u64, duration_s: DurationS): u64 {
    let yearly = math::mul_div_u128(
        reward_amount as u128,
        constants::seconds_per_year() as u128,
        duration_s.raw() as u128,
    );
    assert!(yearly <= (std::u64::max_value!() as u128), errors::max_rate());
    // A budget too thin to annualize would install a campaign that pays nothing all window,
    // with no error and no event.
    assert!(yearly != 0, errors::invalid_params());
    yearly as u64
}
```

Remediation

Implement the above-mentioned suggestions.

Patch

1. The argument is no longer redundant due to the fix in OS-FLR-SUG-02 (issue #1).
2. Issue #2 above was resolved in 692e5d7.
3. Fluid acknowledged this suggestion, stating that both operands are `u64`-sourced, so their product can never approach the `u128` bound.

© 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.