

Security Assessment Report



Fluid Lending & Rewards

September 2026

Prepared for Fluid / Mysten Labs

Table of Contents

Project Summary **3**
- Project Scope 3
- Project Overview 3
- Protocol Overview 3

Assessment Methodology **5**

Threat and Security Overview **6**
- Findings Summary 9
- Severity Matrix 9

Detailed Findings **10**
- Low Severity Issues 12
  - L-01: Withdrawals and redemptions can socialize Liquidity rounding losses to remaining holders 12
- Informational Severity Issues 15
  - I-01: create_manager accepts an obsolete writer cap and creates a permanently unusable manager 15
  - I-02: revoke_rewards_writer accepts redundant revocations and emits configuration event 16
  - I-03: create_manager and install_liquidity_cap lack event emissions 17
  - I-04: can_operate state-dependently aborts for unsupported Borrow and Payback actions 18
  - I-05: RateAdminRole can be managed but grants no authority 20
  - I-06: Unfunded reward liabilities accrue Liquidity yield through TEP 21
  - I-07: MetadataCap<FToken<T>> is permanently entombed in LendingState, freezing fToken metadata 23
  - I-08: RewardsAuth authority handover is single-step, irreversible, and simultaneously strips the outgoing seat's operator rights 24
  - I-09: rebalance can purchase one unnecessary raw unit by converting the rounded asset gap 25

Disclaimer **27**

About Certora **27**

certora

Project Summary

Project Scope

Project Name	Repository Link	Commit Hash	Platform
Fluid Lending & Rewards	https://github.com/Instadapp/fluid-packages-sui	Initial Lending: b4dd97 Initial Rewards: 29befd Final: 838fd2b	Sui / Move

Project Overview

This document describes the security review of **Fluid Lending & Rewards**. The work was undertaken from **August 20th, 2026** to **September 2nd, 2026**.

The following contract list is included in our scope:

packages/lending/sources/**/*.move
packages/lending_rewards/sources/**/*.move
packages/library/sources/**/*.move

The team performed a manual audit of all the files in scope**.** During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Protocol Overview

Lending is Fluid's yield vault on the Liquidity Layer. Users deposit an underlying asset and receive a transferable fToken. The vault only supplies into the shared pool (it never borrows); interest comes from liquidity's supply exchange price. A second token exchange price is the fToken share price: liquidity yield plus any live rewards. Deposit, mint, withdraw, and redeem convert shares against that price. Admins list vaults and bind each to a liquidity reserve. When rewards lift the share price without a matching deposit, a rebalancer tops up the liquidity position so fToken NAV

stays backed.

Lending rewards plans those incentive campaigns. It does not hold or pay a separate reward token; yield is folded into the fToken price. The schedule lives on the lending vault. This package seals the writer cap, checks planner authority, and starts, stops, queues, or cancels windows (budget annualized over a duration, optional follow-up). Promoting a queued window after the current one ends is permissionless. Lending still accrues and rebalances; this package only decides the campaign.

Assessment Methodology

Our assessment approach combines design level analysis with a deep review of the implementation to ensure that a protocol is secure, economically sound, and behaves as intended under realistic conditions.

At the design level, we evaluate the architecture, the economic assumptions behind the protocol, and the safety properties that should hold independently of a specific chain or environment. This process includes reviewing internal and cross protocol interactions, state transition flows, trust boundaries, and any mechanism that could be exploited to extract value, deny service, or alter core system behavior. At this stage, a focused threat modelling exercise helps identify key attack surfaces and adversarial capabilities relevant to the system. Design level issues often relate to incentive structures, governance implications, or systemic behavior that emerges under adversarial conditions.

Implementation analysis focuses on the concrete behavior of the code within the execution model of the target chain. This involves reviewing the correctness of logic, access control, state handling, arithmetic behavior, and the nuanced behaviors of the chain environment. Familiar classes of vulnerabilities such as reentrancy conditions, faulty permission checks, precision issues, or unsafe assumptions often surface at this layer. These findings require context aware reasoning that takes into account both the code and the architectural intent.

To support this analysis, the codebase is examined through repeated manual passes and supplemented by automated tools when appropriate. High-risk logic areas receive deeper scrutiny, invariants are validated against both design intent and actual implementation, and potential vulnerability leads are thoroughly investigated. Automated techniques such as static analysis, fuzzing, or symbolic execution may be used to complement manual review and provide additional insight.

Collaboration with the development team plays an important role throughout the audit. This helps confirm expected behaviors, clarify design assumptions, and ensure an accurate understanding of the protocol's intended operation. All findings are documented with clear reasoning, reproducible examples, and actionable recommendations. A follow up review is conducted to validate the applied fixes and verify that no regressions or secondary issues have been introduced.

Threat and Security Overview

Lending and lending rewards together form Fluid's fToken yield vault on the Liquidity Layer. Each `LendingState<T>` mints transferable `FToken<T>` shares, supplies the underlying through a bound `UserCap<T>`, and prices those shares with a token exchange price that compounds liquidity yield plus any live rewards schedule. End users deposit, mint, withdraw, and redeem against the vault. Campaign policy is planned in `fluid_lending_rewards` (`RewardsAuth`, `RewardsManager`) and written onto lending state through a `RewardsWriterCap`. Lending does not borrow; solvency depends on the liquidity position remaining able to pay token-exchange-price NAV, with a rebalancer expected to fund the unbacked rewards gap. Its main security boundaries are share-to-backing consistency, capability and writer-cap custody, rewards-schedule privilege, arithmetic and exchange-price consistency, and liquidity-layer availability. The following threat model summarizes the assets, actors, assumptions, and a non-exhaustive list of attack vectors considered during the review.

Assets

- **fToken share claims** - principal and accrued yield, including folded rewards.
- **Vault backing** - the lending holder's with-interest supply position at liquidity.
- **Token-exchange-price state** - share valuation (liquidity yield plus rewards).
- **Liquidity-exchange-price checkpoint** - last settled liquidity supply price on the vault.
- **Rewards schedule** - current and queued windows, TVL floor, and yearly budgets.
- **RewardsWriterCap** - authority to install or clear a schedule on a vault.
- **RewardsManager binding** - sealed writer cap tying planners to one vault.
- **RewardsAuth configuration** - authority, planners, version, and upgrades.
- **Lending Auth and roles** - admins, rebalancer, and rate-admin grants.
- **LendingRegistry** - version gate and per-token vault identity.
- **Rebalance funding** - assets used to close NAV above backing.

Actors

- **Lending governance** - lists vaults, manages admins and roles, mints or revokes writer caps, migrates.
- **Rebalancers** - supply extra underlying into the holder position without minting shares.

- **Rewards authority** - manages the planner set and migrates rewards auth.
- **Rewards planners** - start, stop, queue, and cancel campaigns through a manager.
- **fToken holders** - deposit, mint, withdraw, and redeem.
- **Permissionless keepers** - `update_rate` and `transition_to_next_rewards`.
- **Liquidity governance and operators** - pause, mode, limits, and UserCap lifecycle on the holder.
- **Coin issuers**

Trust Assumptions

- **Liquidity correctness** - reserve accounting, limits, pauses, and `operate` semantics hold.
- **UserCap seal** - the holder cap is installed on `LendingState` and not used off-vault.
- **With-interest binding** - the vault remains an active with-interest supplier.
- **Rebalance liveness** - unbacked rewards NAV is funded before exits exhaust backing.
- **Writer-cap custody** - unused or reminted caps are not used to bypass the planner.
- **Conservative campaigns** - duration, TVL floor, and yearly reward stay within intended bounds.
- **Token availability** - listed coins remain transferable and issuer-supported.
- **Sui correctness** - object, coin, clock, and transaction semantics hold.
- **Update liveness** - rates and queued windows are settled in time.
- **Upgrade integrity** - package upgrades preserve state and reviewed security invariants.

Attack Vectors

- **Unbacked rewards accrual** - token exchange price rises without a matching liquidity deposit.
- **Redemption race** - early exits take TEP-priced assets from later holders' principal.
- **Rebalance shortfall** - partial, paused, or mis-priced top-ups leave NAV unbacked.
- **Writer-cap bypass** - lending admin remint or a loose cap writes a schedule without RewardsAuth.
- **Manager desync** - sealed cap and live `rewards_writer` diverge after remint or revoke.
- **Queued-window races** - stop, start, or permissionless transition pays the wrong interval.
- **Spot-TVL rate** - a low-TVL snapshot applies a high rate to an interval that had more deposits.
- **Liquidity-limit stranding** - withdrawal floors, decay, utilization, or `min_operate` block exits while TEP still rises.

- **Binding freeze** - pause, interest-free mode, or cap revoke traps shares while `update_rate` continues.
- **UserCap install race** - first-write or off-vault operate before the cap is sealed on state.
- **Share-conversion overflow** - TEP × shares exceeds u64 and freezes deposit, exit, and rebalance.
- **Rounding leakage** - floor/ceil mint, redeem, and liquidity raw conversion extract or strand dust.
- **Preview/execute price split** - mint sizes payment on a pre-operate TEP and issues shares after LEP moves.
- **Exchange-price truncation** - simulated LEP or accrued TEP wraps on downcast and aborts or misprices.
- **Privileged-role compromise** - halt exits, install max-rate campaigns, or withhold rebalance.
- **Liquidity contagion** - reserve insolvency, revenue skim, or shared-pool stress blocks fToken redeem.
- **Regulated-coin controls** - issuers pause or deny-list the underlying used by the vault.
- **State-observability gaps** - stale TEP/LEP or schedule views mislead integrators and keepers.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	-	-	-
Medium	-	-	-
Low	1	1	1
Informational	9	9	6
Total	10	10	7

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
L-01	Withdrawals and redemptions can socialize Liquidity rounding losses to remaining holders	Low	Fixed
I-01	create_manager accepts an obsolete writer cap and creates a permanently unusable manager	Informational	Fixed
I-02	revoke_rewards_writer accepts redundant revocations and emits configuration event	Informational	Fixed
I-03	create_manager and install_liquidity_cap lack event emissions	Informational	Fixed
I-04	can_operate state-dependently aborts for unsupported Borrow and Payback actions	Informational	Fixed
I-05	RateAdminRole can be managed but grants no authority	Informational	Fixed
I-06	Unfunded reward liabilities accrue Liquidity yield through TEP	Informational	Acknowledged
I-07	MetadataCap<FToken<T>> is permanently entombed in LendingState, freezing fToken metadata	Informational	Acknowledged

I-08	RewardsAuth authority handover is single-step, irreversible, and simultaneously strips the outgoing seat's operator rights	Informational	Fixed
I-09	rebalance can purchase one unnecessary raw unit by converting the rounded asset gap	Informational	Acknowledged

Low Severity Issues

L-01: Withdrawals and redemptions can socialize Liquidity rounding losses to remaining holders

Severity: Low	Impact: Low	Likelihood: Medium
Files: packages/lending/sources/module/user.move	Status: Fixed	

Description:

Lending prices exits using the fToken exchange price (`TEP`), while Liquidity independently converts the asset withdrawal into raw supply units using the supply exchange price (`SEP`).

For `withdraw`, Lending burns:

```
ceil(assets * 1e12 / TEP)
```

shares. For `redeem`, it returns:

```
floor(shares * TEP / 1e12)
```

assets. Liquidity subsequently removes:

```
ceil(assets * 1e12 / SEP)
```

raw units.

When these conversions do not align, Liquidity can remove more backing than the NAV extinguished by the burned shares. The exiting holder receives only the requested assets, but the additional rounding loss is left with the outstanding fTokens. Splitting an exit into repeated small operations allows this loss to accumulate and may prevent remaining holders from withdrawing or redeeming their advertised NAV.

Assume `TEP = 1.0` and `SEP = 1.1`. Alice deposits 1,100 assets and Bob deposits 110, leaving 1,210 NAV backed by 1,100 raw units worth 1,210 assets. For each 10-asset exit:

```
Shares burned = ceil(10 / 1.0) = 10
Raw removed   = ceil(10 / 1.1) = 10
Backing lost  = 10 * 1.1       = 11
```

Bob can repeat the exit eleven times and recover his full 110 assets, while Alice's remaining 1,100 shares are backed by only 1,089 assets. Splitting Bob's shares into eleven 10-share redemptions produces the same result.

Also, this proof of concept demonstrates both paths and shows that the remaining holder cannot fully exit. It can be executed with:

```
cd packages/lending
sui move test withdraw_rounding_poc
```

Recommendations:

For `withdraw`, preserve the requested asset payout but calculate the exact backing removed:

```
raw_removed = ceil(assets * 1e12 / SEP)

backing_removed =
    floor(raw_before * SEP / 1e12)
    - floor((raw_before - raw_removed) * SEP / 1e12)

shares_to_burn = ceil(backing_removed * 1e12 / TEP)
```

For `redeem`, preserve the supplied share burn but reduce the asset payout:

```
nav = floor(shares * TEP / 1e12)
max_raw = floor(nav * 1e12 / SEP)
assets = floor(max_raw * SEP / 1e12)
```

Update `preview_withdraw` and `preview_redeem` to use the same calculations and ensure that every exit removes no more backing than the NAV liability it extinguishes.

Customer Response:

Fixed in commit 00cddb, 1e252a8 & 9fd286b.

Fix Review:

Fixed.

Informational Severity Issues

I-01: create_manager accepts an obsolete writer cap and creates a permanently unusable manager

Files:

packages/lending_rewards/sources/module/admin.move

Description:

`create_manager` verifies that the supplied `RewardsWriterCap` was minted for the provided lending state, but it does not verify that the cap is still the state's currently configured rewards writer. Consequently, an authorized operator can wrap a cap that has already been displaced or revoked, creating a shared `RewardsManager` whose campaign operations will always revert.

The manager cannot be repaired because its cap has no extraction or replacement path. While this does not affect the active writer or protocol funds, it creates an unnecessary zombie manager and delays detection of an operator configuration error until a later campaign operation. This also adds another manager that off-chain systems must distinguish from the vault's live manager.

Recommendations:

Consider validating the active binding before sharing the manager:

```
assert!(
    vault.rewards_writer() == option::some(object::id(&cap)),
    errors::invalid_params(),
);
```

Customer Response:

Fixed in commit 00cddb.

Fix Review:

Fixed.

I-02: revoke_rewards_writer accepts redundant revocations and emits configuration event

Files:

packages/lending/sources/module/admin.move

Description:

The `revoke_rewards_writer` function does not check whether a rewards writer is currently configured.

If `state.rewards_writer()` is already `None` and there is no rewards schedule, the call succeeds, writes `None` over the existing `None`, and emits `LogSetRewardsWriter(..., None)`. As a result, event consumers cannot distinguish an actual writer revocation from a redundant call that did not change the state.

This behavior is also inconsistent with most other authorization-management functions in the codebase, which explicitly reject redundant grants and revocations so that no-op governance transactions are surfaced instead of silently accepted.

Recommendations:

Consider checking that a writer is installed before clearing it:

```
assert!(
    state.rewards_writer().is_some(),
    errors::rewards_writer_not_configured(),
);
```

Customer Response:

Fixed in commit 00cddb.

Fix Review:

Fixed.

I-03: create_manager and install_liquidity_cap lack event emissions

Files:

packages/lending_rewards/sources/module/admin.move
packages/lending/sources/state/lending_state.move

Description:

`create_manager` wraps a `RewardsWriterCap` into a shared `RewardsManager` and does not emit an event. `install_liquidity_cap` stores the liquidity `UserCap` on the vault and does not emit an event.

The cap mints already emit (`LogSetRewardsWriter`, `LogMintUserCap`), so indexers can see that a cap was created but not which object it was bound to. Impact is off-chain tracking only.

Recommendations:

Emit an event in `create_manager` with the manager, vault, and cap ids. Emit an event in `install_liquidity_cap` with the vault and cap ids.

Customer Response:

Fixed in commit 00cddb.

Fix Review:

Fixed.

I-04: can_operate state-dependently aborts for unsupported Borrow and Payback actions

Files:

packages/lending/sources/helpers/accounting.move

Description:

`can_operate` accepts Liquidity's general `Action` type, which includes `Supply`, `Withdraw`, `Borrow`, and `Payback`. However, Lending can only perform supply-side operations.

The function is documented as returning whether the requested action can currently execute. Nevertheless, when the vault has an active Liquidity binding, passing `Borrow` or `Payback` reaches `get_user_supply_pause`, whose supply-position implementation aborts with `10006` for borrow-side actions.

Consequently, the same query behaves differently depending on unrelated vault state:

- Before the Liquidity binding becomes active, it returns `false`.
- After activation, it aborts with `10006`.

This makes `can_operate` unsafe to treat as a general capability predicate. Integrators must know its undocumented action subset and filter inputs themselves.

Recommendations:

Consider returning `false` for unsupported Lending actions before querying the supply-position pause state:

```
if (!has_liquidity_binding(state, reserve, liq_registry)) {
    return false
};

if (
    action == pause::action_borrow() ||
    action == pause::action_payback()
) {
    return false
};
```

Customer Response:

Fixed in commit 00cddb.

Fix Review:

Fixed.

I-05: RateAdminRole can be managed but grants no authority

Files:

packages/lending/sources/state/auth.move

Description:

`RateAdminRole` is exposed as a supported authorization role. Administrators can grant and revoke it through `admin::update_role`, which updates the stored authorization state and emits a `LogRoleChange` event. The role is also included in the package's tests and fixtures.

However, no production entry point checks `RateAdminRole`. In particular, `user::update_rate` is intentionally permissionless, while `RebalancerRole` is the only operator role enforced by the lending package.

Consequently, granting or revoking `RateAdminRole` succeeds and emits an event without changing the holder's executable permissions. Although this does not create an access-control vulnerability, it leaves a misleading governance surface: operators, indexers, and future maintainers may interpret these transactions and events as meaningful permission changes when they have no effect.

Recommendations:

Remove `RateAdminRole` and its support from the role-management dispatcher, together with its related tests and fixtures. The existing role-management functionality should be retained only for roles that are actually enforced.

Customer Response:

Fixed in commit 00cddb & f90e36.

Fix Review:

Fixed.

I-06: Unfunded reward liabilities accrue Liquidity yield through TEP

Files:

packages/lending/sources/helpers/accounting.move

Description:

Reward accrual increases TEP before the corresponding assets are supplied through `rebalance`. Subsequent rate updates calculate Liquidity yield from the complete previous TEP:

```
TEP_new = TEP_old + TEP_old * (reward_return + liquidity_return)
```

Because previously accrued rewards are already included in `TEP_old`, they receive subsequent SEP growth as though they had been supplied to Liquidity. If no rebalance occurred, this yield was not actually earned and instead becomes an additional liability that the rebalancer must fund.

Consider a vault with `1,000` shares and `1,000` assets of backing. A reward schedule distributes `100` assets over one year, while SEP increases by 10% during each half-year. After the first half:

```
Liquidity yield: 1,000 * 10% = 100
Rewards accrued:                  50

NAV:     1,150
Backing: 1,100
Gap:        50
```

Without rebalancing, the second rate update applies the next 10% Liquidity return to the complete `1,150` NAV:

```
Liquidity increase: 1,150 * 10% = 115
New rewards:                         50

NAV:     1,315
Backing: 1,210
Gap:       105
```

Only `110` of the Liquidity increase was earned by the `1,100` assets actually supplied. The remaining `5` is yield credited to the unfunded `50` reward liability:

```
50 * 10% = 5
```

Therefore, the unfunded gap grows from the scheduled `100` rewards to `105`. If the first `50` had been rebalanced halfway through the campaign, that amount would have been supplied and its additional `5` of yield would be legitimate.

This behavior makes the final funding requirement depend on SEP growth, checkpoint frequency, and rebalance timing. If the rebalancer does not cover the increased gap, the vault remains underbacked and later withdrawals may fail. The same accounting model is present in the EVM and Solana implementations, suggesting an implicit assumption that accrued rewards are rebalanced promptly.

Recommendations:

Document that accrued rewards begin receiving SEP growth immediately after being included in TEP, regardless of whether they have been funded, and that the rebalancer must cover the resulting dynamic `NAV - backing` gap.

If this behavior is not intended, calculate Liquidity yield from the actual raw-backed position rather than the complete TEP. Alternatively, pre-fund rewards before accruing them into TEP.

Customer Response:

Acknowledged - *"Rewards lift the fToken price with no matching deposit at liquidity, so NAV owed drifts above what the position actually backs; funding that gap is what rebalancing is for. Additionally, lenders should properly be accounted the rewards the moment they earn them, not the moment they are funded. This is acceptable and known drift. I.e. this is the fair version for the user, not the exact reward amount solution, and the risk around that is ok in exchange for putting the user first"*

I-07: MetadataCap<FToken<T>> is permanently entombed in LendingState, freezing fToken metadata

Files:

packages/lending/sources/state/lending_state.move

Description:

`create_internal` finalizes the freshly created `Currency<FToken<T>>` and immediately buries the returned `MetadataCap<FToken<T>>` in a dynamic field under `keys::share_metadata_key()`. This key is referenced nowhere else in the package, meaning there is no way to borrow, remove, or reach the capability. As a result, the fToken's auto-derived symbol, name, description, and icon can never be updated or explicitly frozen by governance, creating an irreversible operational lock on market metadata.

Recommendations:

Expose an admin-gated accessor to borrow the `MetadataCap<FToken<T>>` so governance can correct metadata.

Customer Response:

Acknowledged - *"This is deliberately a design decision consistent with the reference codebase."*

I-08: RewardsAuth authority handover is single-step, irreversible, and simultaneously strips the outgoing seat's operator rights

Files:

packages/lending_rewards/sources/module/admin.move

Description:

`update_authority` transfers the sole governance seat to an arbitrary address in a single step with no proposal/acceptance handshake. Because it simultaneously invokes `remove_auth` on the outgoing authority, an erroneous transfer (e.g., a typo or wrong-chain address) completely locks governance and irreversibly strips the original key of its operational privileges. Furthermore, losing governance permanently blocks future `admin::migrate` upgrades since the old version check will persistently fail.

Recommendations:

Adopt a two-step handover mechanism utilizing `pending_authority` and an `accept_authority` method. Keep the outgoing authority's operator privileges intact until the handover completes successfully.

Customer Response:

Fixed in commit 766473.

Fix Review:

Fixed - *"RewardsAuth now follows the multi-authority model used by Lending."*

I-09: rebalance can purchase one unnecessary raw unit by converting the rounded asset gap

Files:

packages/lending/sources/module/admin.move

Description:

`rebalance` calculates the difference between the fToken NAV and its Liquidity backing:

```
backed = floor(raw_now * SEP / 1e12)
gap    = NAV - backed
```

It then converts this rounded asset gap into the raw units that must be added:

```
raw_delta = ceil(gap * 1e12 / SEP)
target    = ceil(raw_delta * SEP / 1e12)
```

Because `backed` is rounded before calculating `gap`, converting the gap independently can produce one more raw unit than the final position actually requires. The calculation should instead determine the total raw balance necessary to back NAV and subtract the raw balance already present.

Assume:

```
SEP       = 1.5e12
raw_now   = 101

backed    = floor(101 * 1.5) = 151
NAV       = 162

gap       = 11
```

The current calculation produces:

```
raw_delta    = ceil(11 / 1.5) = 8
target       = ceil(8 * 1.5)  = 12

raw_after    = 101 + 8        = 109
backed_after = floor(109 * 1.5) = 163
```

However, only seven additional raw units are required:

```
raw_required = ceil(162 / 1.5) = 108
raw_delta    = 108 - 101       = 7

target       = ceil(7 * 1.5)   = 11
backed_after = floor(108 * 1.5) = 162
```

Both targets exceed Liquidity's minimum operation amount of 10, but the current calculation supplies 12 assets when 11 would fully close the deficit.

The excess remains credited to the LendingState's Liquidity position and may be consumed by later reward accrual. Nevertheless, the rebalancer provides more capital than necessary. At the Liquidity supply-cap boundary, the calculated eight-raw-unit increase may also abort even when sufficient capacity remains for the required seven raw units.

Recommendations:

Calculate the required raw increase from the final raw balance needed to back NAV:

```
raw_required = ceil(NAV * 1e12 / SEP)
raw_delta    = raw_required - raw_now

target       = ceil(raw_delta * SEP / 1e12)
```

Continue asserting that `backed_after >= NAV` whenever the rebalancer provides the full calculated target.

Customer Response:

Acknowledged - *"rebalance is gated to a governance-appointed RebalancerRole topping up the vault's own backing — a rounding-unit of extra capital landing back in that same position isn't a loss to anyone, just a trusted party slightly over-funding its own top-up."*

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline and is widely used by developers and security researchers during audits and bug bounties.

Certora also provides architectural design reviews, where researchers analyze system design and trust boundaries early to identify structural risks and reduce complexity before implementation.

Certora conducts off-chain audits covering backend services, APIs, frontend, and mobile applications, focusing on vulnerabilities in authentication, data handling, key management, and wallet-related workflows.

Certora also offers Penetration Testing and Red Teaming to simulate real-world attacks and uncover exploitable weaknesses across infrastructure, applications, and business logic.

In addition, Certora provides operational security (OpSec) assessments, reviewing key management, access controls, and operational processes to strengthen overall security posture.