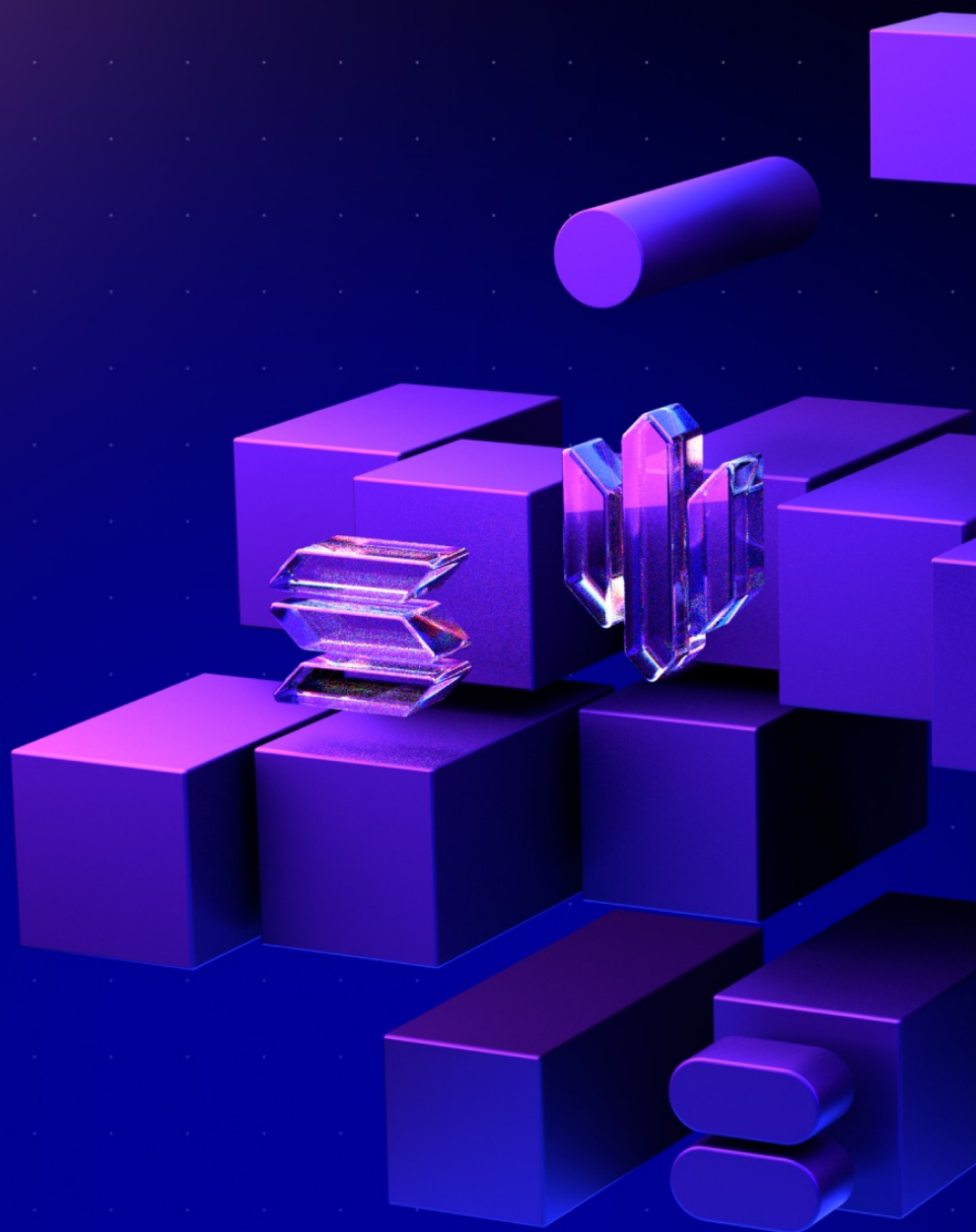
ackee
blockchain security

Kamino

Kamino Lending

18.5.2026



Audit Report

Contents

1. Document Revisions . . . 3
2. Overview . . . 4
   2.1. Ackee Blockchain Security . . . 4
   2.2. Audit Methodology . . . 5
   2.3. Finding Classification . . . 6
   2.4. Review Team . . . 8
   2.5. Disclaimer . . . 8
3. Executive Summary . . . 9
   Revision 1.0 . . . 9
   Revision 1.1 . . . 11
4. Findings Summary . . . 13
Report Revision 1.0 . . . 16
   Revision Team . . . 16
   System Overview . . . 16
   Trust Model . . . 17
   Findings . . . 18
Report Revision 1.1 . . . 64
   Revision Team . . . 64
Appendix A: How to cite . . . 65

1. Document Revisions

1.0	Final Report	07.09.2025
1.1	Fix Review	18.05.2026

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

The Ackee Blockchain Security auditing process follows a routine series of steps:

1. **Code review**
    a. High-level review of the specifications, sources, and instructions provided to us to make sure we understand the project's size, scope, and functionality.
    b. Detailed manual code review, which is the process of reading the source code line-by-line to identify potential vulnerabilities. We focus mainly on common classes of Solana program vulnerabilities, such as:

       missing ownership checks, missing signer authorization, signed CPI of unverified programs, cosplay of Solana accounts, missing rent exemption assertion, bump seed canonicalization, incorrect accounts closing, casting truncation, numerical precision errors, arithmetic overflows or underflows.

    c. Comparison of the code and given specifications, ensuring that the program logic correctly implements everything intended.
    d. Review of best practices to improve efficiency, clarity, and maintainability.
2. **Testing and automated analysis**
    a. Run client's tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests using our testing framework Trident.
3. **Local deployment + hacking**
    a. The programs are deployed locally, and we try to attack the system and break it. There is no specific strategy here, and each project's attack attempts are unique to its implementation.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Andrej Lukačovič	Lead Auditor
Felipe Donato	Auditor
Matej Hyčko	Auditor
Max Kupchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Kamino is a protocol that allows lending and borrowing of assets on Solana Blockchain.

Revision 1.0

Kamino engaged Ackee Blockchain Security to perform a security review of Kamino Lending with a total time donation of 29 engineering days in a period between August 13 and September 12, 2025, with Andrej Lukačovič as the lead auditor.

The audit was performed on the commit `1729408`[1] in the contracts repository.

We began our review by getting familiar with the protocol and its documentation. In the initial stage of the audit, we paid attention to the shallow details such as the protocol's design, instructions, and main business logic.

In the second stage, we started a deeper dive into the program logic, how it was handled, and the implementation details. During this stage, we paid special attention to the following areas:

- ensuring the implementation was overall correct and there was no gap between what was implemented and what was intended;
- ensuring the implementation was secure and there were no potential access control issues;
- looking for common issues as well as identifying potential vulnerable spots that could cause a significant impact on the protocol if misused;
- looking for mathematical issues such as overflows, underflows, and precision errors; and
- ensuring all possible branches and paths were implemented correctly due

to the numerous configuration fields.

In the final stage of the audit, we concluded our findings and prepared the report.

Our review resulted in 18 findings, ranging from Info to High severity.

During the audit, we found 1 High severity issue which we described as a Logic error. Described in H1, the issue arose from the fact that the protocol is "comparing apples and oranges". During liquidation, the collateral amount is withdrawn by the liquidator to compensate for their amount repaid for the loan. The collateral to be withdrawn should be compared to the maximum available collateral amount that can be withdrawn (based on liquidity available in the reserve). However, the instruction does not convert available liquidity but instead compares liquidity available in the reserve to the collateral amount that should be withdrawn. This issue can potentially lead to a scenario where part of the liquidity is withdrawn even though it is meant for withdrawal of users queued for withdrawal.

Another issue that we spotted is around the referral system implemented. This feature appeared incomplete or not fully finished. The issues L3, L5, and L6 describe flaws in this feature. In principle, there are no time/volume-based gated rewards, so referrers can immediately collect rewards. This is tied to the fact that anyone can set themselves as a self-referrer, allowing users to freely collect referrer rewards using the self-referrer mechanism. We understand that the full resolution would likely involve more sophisticated mitigations, such as KYC, but we concluded that this approach is insufficient to protect the protocol from potentially losing its fees.

Last but not least, findings L1, M1, and L4 describe issues with validations of mint accounts (tokens) that are accepted as liquidity tokens. More precisely, the `PermanentDelegate` extension allows the delegate to burn or transfer tokens at will, which can lead to draining of liquidity reserves if misused. In

terms of Token Extension validations, even though the protocol validates that Transfer Fees are zero, it does not validate whether the fees can be updated. This can result in potential Denial of Service when fees are initially zero but updated later, which would prevent users from withdrawing their assets.

We understand that the protocol wants to support a range of tokens, which in some cases require Permanent Delegate due to some jurisdictions. In this case, the protocol needs to pay special attention to allowing such tokens into the system, and the Permanent Delegate needs to be correctly handled by a multisig wallet.

Ackee Blockchain Security recommends Kamino:

- read and review the complete audit report;
- address all identified issues; and
- discuss potential mitigations for the identified issues if required.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Kamino engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

The review was performed between September 23 and May 18, 2025.

Kamino provided multiple commits that addressed the reported findings.

The issue H1 was fixed in commit `c61808a`[2] by correctly evaluating the `max_redeemable_collateral` by calculating total available redeemable collateral amount.

The issue M3 was fixed in commit `47d67e8`[3] by removing the entire

`UpdateEntireReserveConfig` option.

The issue W1 was fixed in commit `52de3d0`[4] by removing the zero lamport check and verifying the account owner and the data field.

The issue W2 was fixed in commit `52de3d0`[5] by checking that the account is either `system_program` owned (Fixable) or `token_program` owned (Valid).

From the reported 18 findings:

- 4 issues were fixed;
- 4 issues were acknowledged; and
- 10 issues remained unresolved.

No new findings were reported.

[1] full commit hash: `1729408323e1de5fed041ec4e6300a23a01fa2b6`

[2] full commit hash: `c61808ae2a2dd349c5ffdc1900cf3851b2b0375b`

[3] full commit hash: `47d67e88daf20f88a14d7e00a64894fc10bf69fb`

[4] full commit hash: `52de3d04063e53313b914e672b0e72dc37106cdc`

[5] full commit hash: `52de3d04063e53313b914e672b0e72dc37106cdc`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	1	3	7	5	2	18

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
H1: Liquidation compares collateral tokens with underlying tokens without exchange rate conversion	High	1.0	Fixed
M1: Time-of-check-time-of-use flaw in token extension handling	Medium	1.0	Acknowledged
M2: Fee evasion through strategic liquidation timing	Medium	1.0	Acknowledged
M3: Privilege escalation via UpdateEntireReserveConfig	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
L1: Inadequate access control for tokens with permanent delegate extension	Low	1.0	Acknowledged
L2: Conflicting liquidation and socialization requirements for zero threshold assets	Low	1.0	Reported
L3: PDA Squatting	Low	1.0	Reported
L4: Lack of check for mint_authority or freeze_authority during Reserve initialization	Low	1.0	Acknowledged
L5: Self-Referral Rebate Abuse in Referrer System	Low	1.0	Reported
L6: Referral Fee Hijacking/Theft	Low	1.0	Reported
L7: Loss of Precision in Borrow Operation	Low	1.0	Reported
W1: Insufficient token account fixability validation in withdrawal queue	Warning	1.0	Fixed
W2: Token account validation bypasses program owner checks	Warning	1.0	Fixed

Finding title	Severity	Reported	Status
W3: Withdrawal queue system lacks cancellation mechanism enabling administrative DoS	Warning	1.0	Reported
W4: Missing Minimum Notice Period for Debt Maturity Allows Instant Liquidation	Warning	1.0	Reported
W5: Lack of validation in LendingMarket name allows duplicates/impersonation confusion	Warning	1.0	Reported
I1: Collateral vault persists after complete withdrawal recovery	Info	1.0	Reported
I2: UserMetadata: Unused and Unvalidated PDA field	Info	1.0	Reported

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Felipe Donato	Auditor
Matej Hyčko	Auditor
Max Kupchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

Kamino is a lending and borrowing protocol on Solana that enables users to supply assets to earn yield and borrow against their collateral. It organizes liquidity into lending markets, each containing multiple reserves (one per supported asset). Users deposit assets into reserves to receive collateral receipt tokens representing their pool share, then use those tokens as collateral in obligations to borrow other assets. Each obligation can hold up to 8 collateral positions and 5 borrows, with health tracked via configurable loan-to-value ratios, liquidation thresholds, and borrow factors. The protocol sources prices from multiple oracle backends (Pyth, Switchboard, Scope) with staleness and confidence validation. Interest accrues to depositors via a configurable borrow rate curve per reserve, with a protocol take rate skimming a portion of interest as fees. Elevation groups provide structured borrowing modes (analogous to e-mode) that modify risk parameters for specific collateral-debt pairs. Undercollateralized positions can be liquidated by third parties who repay debt in exchange for discounted collateral, with configurable close factors, liquidation bonuses, and protocol liquidation fees. An auto-deleveraging mechanism allows the risk council to mark obligations for forced deleveraging during extreme conditions. The program also

supports flash loans (uncollateralized single-transaction borrows with CPI restrictions), a withdraw queue with tickets for when reserve liquidity is insufficient, obligation and borrow orders for automated position management, a referral program that shares protocol fees with referrers, and integration hooks to external farming programs for incentive distribution. Administrative control is layered across a global admin (protocol-wide fee parameters), lending market owners (market and reserve configuration), a risk council (loss socialization and deleveraging), and a proposer authority (reserve initialization). Markets can be made immutable, and an emergency mode can halt most user-facing instructions.

Trust Model

The Kamino Lending program implements a pooled lending protocol with multiple layers of administrative control.

In the Kamino Lending program, users must trust that:

- the Lending Market Owner will not manipulate market parameters (emergency mode, borrow/liquidation limits, elevation groups, insolvency risk thresholds) to disadvantage depositors or borrowers;
- the Lending Market Owner will not set the market to emergency mode, which disables most user-facing instructions including deposits, withdrawals, borrows, repayments, and liquidations;
- the Lending Market Owner will not misconfigure reserve parameters (LTV ratios, liquidation thresholds, borrow rate curves, deposit/borrow caps, oracle settings) in ways that expose the protocol to bad debt or unfair liquidations;
- the Lending Market Owner will not abuse the two-step owner transfer mechanism to hand control to a malicious party;
- the Risk Council will invoke `socialize_loss` only when genuinely needed to

distribute bad debt across lenders, and will not abuse `mark_obligation_for_deleveraging` to force-liquidate healthy positions;

- the Proposer Authority will not initialize reserves with misconfigured parameters during the pre-activation setup phase;
- the Global Admin will not set inappropriate protocol fee parameters (protocol take rate, protocol liquidation fee, origination fees, flash loan fees) that extract excessive value from users;
- the Global Admin will not transfer admin rights to a malicious party via the pending admin mechanism;
- the Fee Collector address is set to a legitimate recipient of protocol fees;
- the configured oracles (Pyth, Switchboard, Scope) provide accurate and timely price data, and that oracle staleness and confidence checks are configured with appropriate thresholds;
- the CPI whitelist (hardcoded integrator program IDs for Squads, Meteora, Kamino Vault, and others) only includes trusted programs, as whitelisted programs can invoke lending instructions via CPI; and
- the immutability flag, once set on a lending market, correctly prevents further market and reserve configuration changes (except global-admin-only fee parameters).

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

H1: Liquidation compares collateral tokens with underlying tokens without exchange rate conversion

High severity issue

Impact:	High	Likelihood:	Medium
Target:	-	Type:	Logic error

Description

The liquidation process incorrectly compares collateral tokens (cTokens) with underlying liquidity tokens without applying the necessary exchange rate conversion. In the `liquidate_obligation` function, the `withdraw_amount` represents collateral tokens while `max_redeemable_collateral` represents underlying liquidity tokens, making the comparison between different token types invalid when the exchange rate is not 1:1.

The following code snippet shows how the `withdraw_collateral_amount` is calculated. At the end of the block, it can be seen that the `withdraw_amount` is compared with the `max_redeemable_collateral`. However, the `max_redeemable_collateral` is in the form of the underlying liquidity tokens, not the collateral tokens.

Listing 1. Excerpt from lending_operations

```
1638 let withdraw_collateral_amount = {
1639     refresh_reserve(
1640         &mut withdraw_reserve_ref_mut,
1641         clock,
1642         None,
1643         lending_market.referral_fee_bps,
1644     )?;
1645     let redeem_collateral_options =
     RedeemCollateralOptions::resolve(liquidation_reason);
1646     let max_redeemable_collateral = if
```

```
    redeem_collateral_options.use_withdraw_queue {
1647            withdraw_reserve_ref_mut.total_available_liquidity_amount()
1648        } else {
1649            withdraw_reserve_ref_mut.freely_available_liquidity_amount()
1650        };
1651        min(withdraw_amount, max_redeemable_collateral)
1652 };
```

This issue can cause incorrect withdrawal amounts to be used, potentially leading to situations where collateral is redeemed into underlying liquidity tokens while the liquidity should be considered locked for the queued collateral.

Exploit scenario

Alice is a regular user.

Bob is a liquidator.

1. Alice deposits SOL into the Solana reserve, receiving cSOL collateral.
2. Alice deposits cSOL tokens into the obligation.
3. Alice borrows USDC.
4. The exchange rate between SOL and cSOL appreciates over time as borrowings generate interest, so the exchange rate changes from 1.0 to 1.05, meaning 1 cSOL = 1.05 SOL.
5. Alice becomes liquidatable due to obligation ordering and market conditions.
6. Bob notices Alice's liquidatable obligation and liquidates it.
7. The `max_redeemable_collateral` calculation block flow:
    - `redeem_collateral_options.use_withdraw_queue` is false due to Obligation Order.
    - The `withdraw_reserve_ref_mut.freely_available_liquidity_amount()`

returns amount in SOL, for example 20,000 SOL.

- The `withdraw_amount` is, for example, 19,500 cSOL.
- Due to the exchange rate, 20,000 SOL is approximately 19,047 cSOL.
- The `min(withdraw_amount, max_redeemable_collateral)` is the following: `min(19,500 cSOL, 20,000 SOL)` ⇒ 19,500 cSOL.
- The `withdraw_collateral_amount` becomes 19,500 cSOL, even though the correct amount should be 19,047 cSOL when the exchange rate is taken into account.

8. Based on the `max_redeemable_collateral`, liquidation will redeem 19,500 cSOL instead of 19,047 cSOL, which overlaps with collateral locked for the queue.

Recommendation

Ensure that the `max_redeemable_collateral` calculation is performed using the correct amount of cSOL, not SOL.

Fix 1.1

The issue was fixed by correctly evaluating the `max_redeemable_collateral` by calculating total available redeemable collateral amount.

Go back to Findings Summary

M1: Time-of-check-time-of-use flaw in token extension handling

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	-	Type:	Code quality

Description

The protocol validates token extensions only at reserve initialization and during user operations, but does not account for the fact that many SPL Token 2022 extensions have update authorities that can modify extension parameters after initialization. This creates a time-of-check-time-of-use (TOCTOU) vulnerability where a malicious actor can initialize a reserve with compliant extension settings, attract user deposits, and then modify the extensions to prevent withdrawals or enable other malicious behaviors.

The vulnerability exists because the `validate_liquidity_token_extensions` function is called at multiple points (reserve initialization, deposits, withdrawals, liquidations, etc.) but the protocol assumes extension parameters remain static. However, extensions like `TransferFeeConfig`, `TransferHook`, `Pausable`, and others can be modified by their respective update authorities after the reserve is created and users have deposited funds.

For example, as shown below the protocol validates that `TransferFeeConfig` has zero fees, but the update authority can change it later, causing user withdrawal transactions to fail validation, thus creating a time-of-check-time-of-use (TOCTOU) vulnerability.

Listing 2. Excerpt from constraints

```
88 ExtensionType::TransferFeeConfig => {
89     let ext = mint
```

```
90
   .get_extension::<spl_token_2022::extension::transfer_fee::TransferFeeConfig>(
   )?;
91     if <u16>::from(ext.older_transfer_fee.transfer_fee_basis_points) != 0
92         || <u16>::from(ext.newer_transfer_fee.transfer_fee_basis_points) != 0
93     {
94         xmsg!(
95             "Transfer fee must be 0 for liquidity tokens, got: {:?}",
96             ext
97         );
98         return err!(LendingError::UnsupportedTokenExtension);
99     }
100 }
```

Exploit scenario

Alice is a malicious actor.

Bob is a legitimate user.

1. Alice creates an SPL Token 2022 mint with a `TransferFeeConfig` extension set to zero transfer fees and sets herself as the transfer fee config authority.
2. The protocol initializes a reserve with this mint, which passes validation since the transfer fee is zero.
3. Bob deposits a significant amount of liquidity into the reserve. The transaction succeeds because the extension validation passes with zero fees.
4. Alice uses her update authority to modify the `TransferFeeConfig` extension, setting the transfer fee to a high value (e.g., 50%).
5. Bob attempts to withdraw his liquidity by calling `withdraw_obligation_collateral_and_redeem_reserve_collateral`. The transaction fails during the `validate_liquidity_token_extensions` check because the transfer fee is no longer zero, violating the protocol's validation rules.

6. Bob's funds become effectively locked in the protocol, as any withdrawal attempt fails the extension validation.

Recommendation

Implement additional validation to ensure that extension update authorities are either disabled (set to `None`) or controlled by the protocol itself. Add checks for update authorities in the `validate_liquidity_token_extensions` function for all extensions that support updates.

Acknowledgment 1.1

The issue was acknowledged by the client with the following response:

> *We cannot enforce token issuers to revoke their authorities (USDG, PYUSD) and tokens that have malicious actors won't be onboarded. Also, we cannot be the authority, you said we can ensure that the update authorities "are controlled by the protocol itself.*
>
> — Kamino team

Go back to Findings Summary

M2: Fee evasion through strategic liquidation timing

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	-	Type:	Code quality

Description

The protocol calculates and collects liquidation fees only when collateral can be fully redeemed for underlying liquidity during the liquidation process. However, in the remaining cases the protocol fee is not collected:

- the first option occurs if there is zero available liquidity to be redeemed. In this case, the `post_liquidate_redeem` function returns `None` instead of fee amounts, and no fees are collected; and
- the second option occurs if there is non-zero available liquidity to be redeemed (partial redemption). In this case, the protocol fee is calculated only on the redeemable portion, not on the full liquidation amount, resulting in proportionally reduced fee collection.

The following code snippet shows that for the `withdraw_collateral_amount` equal to zero, the protocol fee is not calculated nor stored, and `None` is returned. When the `withdraw_collateral_amount` is non-zero, the protocol fee is calculated and returned.

Listing 3. Excerpt from lending_operations

```
1672 pub(crate) fn post_liquidate_redeem(
1673     withdraw_reserve: &mut Reserve,
1674     repay_amount: u64,
1675     withdraw_amount: u64,
1676     withdraw_collateral_amount: u64,
1677     liquidation_bonus_rate: Fraction,
```

```
1678     liquidation_reason: LiquidationReason,
1679     min_acceptable_received_liquidity_amount: u64,
1680     clock: &Clock,
1681 ) -> Result<Option<(u64, u64)>> {
1682     // if there is liquidity redeem it
1683     if withdraw_collateral_amount != 0 {
1684         let withdraw_liquidity_amount = redeem_reserve_collateral(
1685             withdraw_reserve,
1686             withdraw_collateral_amount,
1687             clock,
1688             RedeemCollateralOptions::resolve(liquidation_reason),
1689         )?;
1690         let protocol_liquidation_fee_pct = match liquidation_reason {
1691             LiquidationReason::LtvExceeded
1692             | LiquidationReason::IndividualDeleveraging
1693             | LiquidationReason::MarketWideDeleveraging
1694             | LiquidationReason::ReserveDebtMaturityReached
1695             | LiquidationReason::ObligationBorrowDebtTermReached => {
1696                 withdraw_reserve.config.protocol_liquidation_fee_pct
1697             }
1698             LiquidationReason::ObligationOrder(_) => {
1699                 withdraw_reserve.config.protocol_order_execution_fee_pct
1700             }
1701         };
1702         let protocol_liquidation_fee =
    liquidation_operations::calculate_protocol_liquidation_fee(
1703             withdraw_liquidity_amount,
1704             liquidation_bonus_rate,
1705             protocol_liquidation_fee_pct,
1706         );
1707         let net_withdraw_liquidity_amount = withdraw_liquidity_amount -
    protocol_liquidation_fee;
1708         xmsg!(
1709             "pnl: Liquidator repaid {} and withdrew {} collateral with fees
    {}",
1710             repay_amount,
1711             net_withdraw_liquidity_amount,
1712             protocol_liquidation_fee
1713         );
1714
1715         if net_withdraw_liquidity_amount <
    min_acceptable_received_liquidity_amount {
1716             return err!(LendingError::LiquidationRewardTooSmall);
1717         }
1718
1719         // Protocol fee is subtracted from withdraw liquidity amount in
```

```
    token transfers
1720        Ok(Some((withdraw_liquidity_amount, protocol_liquidation_fee)))
1721    } else {
1722        let theoretical_withdraw_liquidity_amount = withdraw_reserve
1723            .collateral_exchange_rate()
1724            .collateral_to_liquidity(withdraw_amount);
1725
1726        if theoretical_withdraw_liquidity_amount <
    min_acceptable_received_liquidity_amount {
1727            return err!(LendingError::LiquidationRewardTooSmall);
1728        }
1729
1730        xmsg!(
1731            "pnl: Liquidator repaid {} and withdrew {} ctokens",
1732            repay_amount,
1733            withdraw_amount
1734        );
1735        Ok(None)
1736    }
1737 }
```

However, as shown in the following code snippet, the maximum available collateral amount is calculated as the minimum of the collateral amount and the maximum redeemable collateral amount.

Listing 4. Excerpt from lending_operations

```
1638 let withdraw_collateral_amount = {
1639     refresh_reserve(
1640         &mut withdraw_reserve_ref_mut,
1641         clock,
1642         None,
1643         lending_market.referral_fee_bps,
1644     )?;
1645     let redeem_collateral_options =
    RedeemCollateralOptions::resolve(liquidation_reason);
1646     let max_redeemable_collateral = if
    redeem_collateral_options.use_withdraw_queue {
1647         withdraw_reserve_ref_mut.total_available_liquidity_amount()
1648     } else {
1649         withdraw_reserve_ref_mut.freely_available_liquidity_amount()
1650     };
1651     min(withdraw_amount, max_redeemable_collateral)
```

```
1652 };
```

The issue described above can result in a scenario where fees are not collected during liquidation at all or only partial fees are collected.

Exploit scenario

Alice is a borrower with an undercollateralized obligation.

Bob is a liquidator aware of the issue.

1. Bob identifies Alice's undercollateralized obligation that can be liquidated using a reserve with zero available liquidity for withdrawal.
2. Bob attempts to liquidate the obligation by calling `liquidate_obligation_and_redeem_reserve_collateral`. The liquidation calculation proceeds normally, determining the appropriate repay amount, withdraw amount, and liquidation bonus. However, when the `post_liquidate_redeem` function attempts to redeem the seized collateral, it finds zero available liquidity in the reserve.
3. Since no liquidity can be redeemed, the `post_liquidate_redeem` function returns `None` for the withdrawal amounts, and no protocol liquidation fees are calculated or collected. Bob receives the full amount of seized collateral tokens (cTokens) without paying any liquidation fees to the protocol.

Recommendation

Ensure the protocol fee is collected for all cases of liquidation.

Acknowledgment 1.1

The issue was acknowledged by the client with the following response:

> *This is a known fee by-pass. We have a todo item for this*

> — Kamino team

Go back to Findings Summary

M3: Privilege escalation via UpdateEntireReserveConfig

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	-	Type:	Data validation

Description

The `UpdateEntireReserveConfig` function allows the lending market owner to bypass global admin restrictions and modify protocol-wide fee parameters, breaking safety assumptions in two sensitive fields of the ReserveConfig. This creates an unmitigable griefing vector where a malicious market owner can persistently DoS reserve operations.

The `is_update_reserve_config_mode_global_admin_only` function correctly identifies global-admin-only parameters:

Listing 5. Excerpt from lending_operations

```
3360 match mode {
3361     UpdateConfigMode::UpdateProtocolTakeRate
3362     | UpdateConfigMode::UpdateProtocolLiquidationFee
3363     | UpdateConfigMode::UpdateHostFixedInterestRateBps
3364     | UpdateConfigMode::UpdateProtocolOrderExecutionFee
3365     | UpdateConfigMode::UpdateFeesOriginationFee
3366     | UpdateConfigMode::UpdateFeesFlashLoanFee => true,
```

These parameters are designated as global admin only.

The `UpdateEntireReserveConfig` function incompletely preserves global-admin-only fields when updating reserve configuration:

Listing 6. Excerpt from lending_operations

```
2199 UpdateConfigMode::UpdateEntireReserveConfig => {
```

```
2200     // Warning: All field restored here are expected to be
2201     // the ones to be updated by the global admin see
     `allowed_signer_update_reserve_config`
2202     msg!(
2203         r"Updating entire reserve config, fields `protocol_take_rate_pct`,
2204         `protocol_liquidation_fee_pct`, `protocol_order_execution_fee_pct`
     and
2205         `host_fixed_interest_rate_bps` will remain unchanged"
2206     );
2207     let old_protocol_take_rate_pct = reserve.config.protocol_take_rate_pct;
2208     let old_protocol_liquidation_fee_pct =
     reserve.config.protocol_liquidation_fee_pct;
2209     let old_host_fixed_interest_rate_bps =
     reserve.config.host_fixed_interest_rate_bps;
2210     let old_protocol_order_execution_fee_pct =
2211         reserve.config.protocol_order_execution_fee_pct;
2212
2213     config_items::for_named_field!(&mut reserve.config).set(value)?;
```

However, the function fails to preserve these global-admin-only fields:

- `reserve.config.fees.origination_fee_sf`
- `reserve.config.fees.flash_loan_fee_sf`

Consequently, a market owner can override Global Admin configurations:

- they are part of the ReserveConfig struct;
- the entire struct gets deserialized and set; and
- the fees are not in the list of fields that get restored.

Additionally, the validation logic has limitations. If the Reserve is inactive, both `flash_loan_fee_sf` and `origination_fee_sf` can be set to concerning values. If the Reserve is active, the validation will only catch the `origination_fee_sf`, leaving `flash_loan_fee_sf` open for further abuse.

Listing 7. Excerpt from lending_operations

```
3498 if u128::from(config.fees.origination_fee_sf) >= FRACTION_ONE_SCALED {
3499     msg!("Origination fee must be in range [0, 100%)");
```

```
3500     return err!(LendingError::InvalidConfig);
3501 }
```

The validation only prevents fees >= 100%, but does not prevent high fees like 99.99%.

Exploit scenario

1. Alice, a malicious lending market owner, exploits this vulnerability to persistently disrupt protocol operations. Bob, the global administrator, attempts to maintain proper fee configurations.
2. Alice calls `UpdateEntireReserveConfig` and sets `origination_fee_sf: 99.99%` and `flash_loan_fee_sf: 99.99%`. The transaction succeeds because Alice is the market owner.
3. Bob detects the attack and fixes it by calling `UpdateFeesOriginationFee(0.1%)`. The transaction succeeds because this is an admin-only function.
4. Alice immediately strikes back and calls `UpdateEntireReserveConfig` again, setting fees back to 99.99%. The transaction succeeds.
5. This creates an infinite loop where Bob calls `UpdateFeesOriginationFee(0.1%)` to fix the configuration, but Alice immediately calls `UpdateEntireReserveConfig(fees: 99.99%)` to revert the changes. This pattern continues indefinitely, creating a persistent DoS condition for reserve operations.

Recommendation

Add `fees.origination_fee_sf` and `fees.flash_loan_fee_sf` to the list of global-admin-only fields that are preserved during `UpdateEntireReserveConfig` operations. Update the function to save and restore these fee values alongside the existing protected fields to prevent unauthorized modification by market owners.

Fix 1.1

The issue was fixed by removing the entire `UpdateEntireReserveConfig` option.

Go back to Findings Summary

L1: Inadequate access control for tokens with permanent delegate extension

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	-	Type:	Code quality

Description

The protocol accepts SPL Token 2022 mints with the `PermanentDelegate` extension as valid liquidity tokens without implementing proper validation or restrictions on the permanent delegate's authority. The permanent delegate extension grants unrestricted transfer and burn capabilities to a designated authority, bypassing normal token account ownership controls.

When a reserve is initialized with a mint that has a permanent delegate, the delegate can transfer or burn tokens from any token account of that mint without requiring approval from the account owner. This creates a vulnerability where the permanent delegate can drain liquidity from the protocol's reserve accounts.

The following code snippet shows that the `PermanentDelegate` extension is included in the `VALID_LIQUIDITY_TOKEN_EXTENSIONS` array.

Listing 8. Excerpt from constraints

```
44 const VALID_LIQUIDITY_TOKEN_EXTENSIONS: &[ExtensionType] = &[
45     ExtensionType::ConfidentialTransferFeeConfig,
46     ExtensionType::ConfidentialTransferMint,
47     ExtensionType::MintCloseAuthority,
48     ExtensionType::MetadataPointer,
49     ExtensionType::PermanentDelegate,
50     ExtensionType::TransferFeeConfig,
51     ExtensionType::TokenMetadata,
52     ExtensionType::TransferHook,
53     ExtensionType::DefaultAccountState,
```

```
    ExtensionType::ScaledUiAmount,
    ExtensionType::Pausable,
];
```

Exploit scenario

Scenario 1

Alice is a regular user.

Bob is a mint creator.

Emma is a hacker.

1. Bob creates a new mint with the `PermanentDelegate` extension. The `PermanentDelegate` authority is an incorrectly handled multisig (3/5) where Bob and his team control all 5 signers.
2. The protocol creates a reserve with the mint that has the `PermanentDelegate` extension.
3. Alice deposits liquidity into the reserve.
4. Emma gains possession of the multisig due to incorrect handling of the multisig wallets, for example by gaining access to the private keys.
5. Emma transfers all tokens from the protocol's `reserve_liquidity_supply` account to her own account or directly burns them.
6. Alice attempts to withdraw her liquidity, but the transaction fails due to insufficient tokens in the reserve account.

Scenario 2

Alice is a regular user.

Bob is a hacker.

1. Bob creates a new lending market.
2. Bob initializes a new reserve with the mint that has the `PermanentDelegate` extension within his lending market.
3. Bob tricks Alice into depositing liquidity into the reserve.
4. Bob transfers all tokens from the protocol's `reserve_liquidity_supply` account to his own account or directly burns them.
5. Alice attempts to withdraw her liquidity, but the transaction fails due to insufficient tokens in the reserve account.

Recommendation

Support for mints with the `PermanentDelegate` extension as liquidity tokens is understandable. Some jurisdictions require such extensions to be present for the ability to seize assets. However, these extensions also pose security risks for end users and the protocol itself.

The recommended approach is to:

- always monitor new mints added as liquidity to the system; or
- create a whitelist of mints that are allowed to be used as liquidity tokens, validate new reserves against that list, and update the list manually by the protocol team.

Acknowledgment 1.1

The issue was acknowledged by the client with the following response:

> *we will monitor the existing and new mints and reassess their opsec.*
>
> — Kamino team

Go back to Findings Summary

L2: Conflicting liquidation and socialization requirements for zero threshold assets

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	-	Type:	Code quality

Description

The protocol allows reserves to be configured with zero liquidation threshold (`liquidation_threshold_pct = 0`), which enables users to deposit collateral that cannot be liquidated during normal liquidation processes. However, the bad debt socialization mechanism requires obligations to have no active deposits before socializing losses. This creates a scenario where obligations with zero liquidation threshold collateral cannot have their bad debt socialized, leading to permanent bad debt that cannot be resolved.

The vulnerability exists in two key areas:

- the `liquidate_obligation` function explicitly prevents liquidation of collateral with zero liquidation threshold, returning `CollateralNonLiquidatable` error; and

Listing 9. Excerpt from lending_operations

```
1533 let (_, collateral_liquidation_threshold_pct) =
1534     get_max_ltv_and_liquidation_threshold(&withdraw_reserve_ref,
     elevation_group);
1535
1536 if collateral_liquidation_threshold_pct == 0 {
1537     xmsg!(
1538         "Liquidation threshold of the withdraw reserve is 0 and can't be
     used for liquidation"
1539     );
1540     return err!(LendingError::CollateralNonLiquidatable);
```

```
1541 }
```

- the `socialize_loss` function requires obligations to have no active deposits before proceeding, returning `CannotSocializeObligationWithCollateral` error if any deposits remain.

Listing 10. Excerpt from lending_operations

```
1843 if !obligation.is_active_deposits_empty() {
1844     msg!("Obligation hasn't been fully liquidated!");
1845     return
     Err(LendingError::CannotSocializeObligationWithCollateral.into());
1846 }
```

This creates an impossible situation where collateral with zero liquidation threshold cannot be removed through liquidation, but must be removed before bad debt can be socialized.

Exploit scenario

Alice is a user.

Bob is a liquidator.

Charlie is the risk council.

1. Alice deposits SOL into the Solana reserve, receiving cSOL collateral worth $1000.
2. Alice deposits ETH into the Ether reserve, receiving cETH collateral worth $1000.
3. The Solana reserve has Loan to Value (LTV) and unhealthy LTV set to 0%, and the Ether reserve has LTV 80% and unhealthy LTV 85%.
4. Alice deposits both collateral types (cSOL and cETH) into the same obligation.

5. Based on the value deposited, Alice can borrow assets worth up to $800.
6. The price of the borrowed asset appreciates significantly and quickly, moving Alice's loan to value above 100%.
7. Bob notices this and decides to liquidate Alice's obligation.
8. Bob repays the debt and wants to withdraw the deposited collateral. Bob cannot withdraw the cSOL because this collateral, due to its LTV being 0%, does not back any debt. Bob can withdraw only the cETH collateral.
9. Bob withdraws all the cETH collateral but does not repay the entire debt because Alice's obligation LTV is above 100%, meaning the debt is greater than the collateral.
10. The obligation reaches a state where the deposited collateral is the cSOL collateral not backing any debt and remaining debt that was not repaid during liquidation.
11. The obligation needs to be socialized because it contains bad debt and cannot be liquidated anymore, as liquidation does not allow withdrawal of collateral with zero unhealthy LTV.
12. Charlie cannot socialize the obligation because the obligation still contains deposits.

Recommendation

Ensure the socialization of bad debt is possible for obligations with zero liquidation threshold collateral.

Go back to Findings Summary

L3: PDA Squatting

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	-	Type:	Data validation

Description

The protocol allows users to create referrer short URLs through Program Derived Addresses (PDAs). The `referrer_token_state` can be freely created, enabling attackers to squat on potentially popular referrer short URLs before legitimate users can claim them.

The vulnerability exists in the `InitReferrerStateAndShortUrl` instruction, which uses the following account structure:

Listing 11. Excerpt from handler_init_referrer_state_and_short_url

```
45 #[account(
46     init,
47     seeds = [BASE_SEED_SHORT_URL, short_url.as_bytes()],
48     bump,
49     payer = referrer,
50     space = SHORT_URL_SIZE + 8
51 )]
52 pub referrer_short_url: Account<'info, ShortUrl>,
```

An attacker can exploit this by first calling `InitUserMetadata` and then `InitReferrerStateAndShortUrl` with any desired string, effectively squatting on valuable referrer short URLs.

Exploit scenario

1. Alice, a malicious actor, generates multiple wallets to circumvent any per-wallet limitations.

2. Alice calls `InitUserMetadata` for each wallet to establish the necessary prerequisites.
3. Alice systematically calls `InitReferrerStateAndShortUrl` with a list of valuable short URLs (such as ”vip”, ”premium”, ”bonus”) across her wallets.
4. Bob, a legitimate user or project team member, attempts to claim a valuable short URL but discovers it has already been taken by Alice.
5. Alice now controls valuable referrer short URLs that she can potentially sell or use to mislead users into believing they are associated with official or premium services.

Recommendation

Implement access control mechanisms to prevent unauthorized short URL registration. Implement a whitelist of approved short URLs, require administrative approval for short URL registration, or implement a time-based reservation system that allows legitimate users to claim valuable URLs.

Go back to Findings Summary

L4: Lack of check for mint_authority or freeze_authority during Reserve initialization

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	-	Type:	Data validation

Description

Reserve initialization does not check for `mint_authority` or `freeze_authority`.

`reserve_liquidity_mint` can be frozen at will, leading to DoS and locked funds in operations within the given reserve.

Exploit scenario

1. Alice, a malicious administrator, observes that the reserve initialization process does not validate the mint and freeze authorities of the `reserve_liquidity_mint` token.
2. Alice initializes a new reserve using a token where Alice controls the freeze authority.
3. Alice waits for users to deposit funds into the reserve, then freezes the `reserve_liquidity_mint` token. This action prevents any operations requiring token transfers within the reserve, effectively creating a denial of service condition and locking user funds.
4. Alternatively, Alice uses mint authority to mint excessively, leading to inflation and consequently making positions unhealthy and liquidatable.

Recommendation

Add validation checks during reserve initialization to ensure that the `mint_authority` and `freeze_authority` of `reserve_liquidity_mint` are either set

to None or controlled by trusted entities. Implement proper authority verification before allowing reserve operations to proceed.

Acknowledgment 1.1

The issue was acknowledged by the client with the following response:

> *we cannot really ensure in code that the address is controlled by trusted entities. we have to do our own research off-chain*
>
> — Kamino team

Go back to Findings Summary

L5: Self-Referral Rebate Abuse in Referrer System

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	-	Type:	Data validation

Description

The Kamino lending program allows users to set themselves as their own referrer in the UserMetadata account, enabling referral fees from borrow origination and interest accruals to accrue to their own ReferrerTokenState. This creates a rebate loop where users effectively pay reduced borrow costs, draining protocol subsidies or reserves without economic justification. No checks prevent referrer == user owner, violating the intended "referrals for others" business logic. Combined with loose token state initialization, this leads to unintended fee redirections, precision losses in accruals, and potential infinite rebate farming.

Listing 12. Excerpt from lending_operations

```
324 if let Some(mut referrer_token_state) = referrer_token_state {
325     if lending_market.referral_fee_bps > 0 {
326         add_referrer_fee(
327             borrow_reserve,
328             &mut referrer_token_state,
329             Fraction::from_num(referrer_fee),
330         )?;
331 
332         borrow_reserve.liquidity.total_available_amount += referrer_fee;
333     }
334 }
```

No check for `obligation.referrer != obligation.owner`. Accrues `referrer_fee` to provided `referrer_token_state` (validated but self-allowed). Similar issue in `compound_interest` (`pending_referrer_fees_sf` shift via `refresh_obligation`).

Root enabler: UserMetadata init sets `referrer = referrer_user_metadata.owner` without `!= self` validation.

Exploit scenario

Bob is the protocol administrator.

1. Alice, a user, initializes her UserMetadata account, setting the referrer pubkey to her own wallet pubkey by providing her own metadata as the optional `referrer_user_metadata`.
2. Alice (or anyone) initializes a ReferrerTokenState PDA for her pubkey and the target reserve. Loose access control allows this without restrictions.
3. Alice borrows liquidity through her obligation account. Since `has_referrer()` returns true (referrer set to self), the borrow calculates and accrues `referrer_fee` to her own ReferrerTokenState (`unclaimed_sf`) and the reserve's `accumulated_referrer_fees_sf`.
4. Alice refreshes her obligation via `refresh_obligation`, shifting any pending referral fees from interest accruals (e.g., from `compound_interest`) to her `unclaimed_sf` in the token state.
5. Alice withdraws the accrued fees from her ReferrerTokenState to her associated token account, effectively rebating the referral portion of her borrow costs.
6. Alice repeats the borrow-refresh-withdraw cycle, amplifying rebates on larger or multiple borrows, draining protocol funds. Bob loses revenue to Alice's self-referral loop.

Recommendation

Increase referral security robustness:

- Implement `referrer != owner` validation in both the metadata initialization and the referral fee accrual logic to ensure referral fees only benefit

genuine third-party referrers.

- Add a validation check in the UserMetadata initialization to prevent users from setting themselves as their own referrer.
- Use Web2-based KYC where appropriate for referral economics.
- Require a minimum deposit/stake to participate.
- Put an incentive behind a time-locked or deposit-volume reward.

Go back to Findings Summary

L6: Referral Fee Hijacking/Theft

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	-	Type:	Data validation

Description

The `FlashRepayReserveLiquidity` instruction allows liquidators to attach an arbitrary `referrer_account` and `referrer_token_state` during repayment.

Validation only ensures that the `referrer_token_state` PDA is well-formed for the given referrer and reserve, but does not verify that this referrer is tied to any obligation or that the user taking the loan was introduced by this referrer.

This allows users to redirect referral fees to themselves or third parties, creating a self-rebate mechanism.

Listing 13. Excerpt from handler_flash_repay_reserve_liquidity

```
31 let referrer_token_state_loader = if referrer_account.is_some() {
32     match &ctx.accounts.referrer_token_state {
33         Some(referrer_token_state_loader) => {
34             let referrer_token_state = &mut
   referrer_token_state_loader.load_mut()?;
35 
36             validate_referrer_token_state(
37                 &crate::ID,
38                 referrer_token_state,
39                 referrer_token_state_loader.key(),
40                 reserve.liquidity.mint_pubkey,
41                 referrer_account.as_ref().unwrap().key(),
42                 ctx.accounts.reserve.key(),
43             )?;
44 
45             Some(referrer_token_state_loader)
46         }
```

```
47          None => {
48              msg!("No referrer account provided");
49              None
50          }
51      }
52  } else {
53      None
54  };
```

Exploit scenario

Alice is a malicious actor.

Bob is a legitimate user.

1. Alice opens a flash borrow and repay transaction.
2. During repayment, Alice provides `referrer_account = Alice's own wallet` and `referrer_token_state = PDA derived from Alice + reserve`.
3. The program validates the PDA structure but does not verify that the referrer is a real referrer who introduced Alice to the protocol.
4. Referral fees are credited to Alice herself.
5. Alice pays reduced fees or completely neutralizes them, draining protocol revenue.
6. Bob runs comparable borrow and repay flows without attaching self-referrer accounts and does not obtain the same fee reduction that Alice obtains through the misrouted referrer credits.

Recommendation

Add a requirement that flash loans also contain an obligation account. The referrer can be compared to the referrer from the obligation. Even though this will not fully mitigate the self-referral abuse, the process will be more transparent.

The self-referral abuse needs more sophisticated mitigations, such as KYC or volume/time-based rewards.

Go back to Findings Summary

L7: Loss of Precision in Borrow Operation

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	-	Type:	Logic error

Description

The borrow operation suffers from precision loss that creates phantom debt accumulation and referrer fee rounding bias.

In the `calculate_borrow_all_available` function, the fractional borrow amount is rounded down when determining the actual withdrawal amount, but the full fractional amount is recorded as borrowed debt. This discrepancy results in phantom debt where users are charged for amounts they never received. Additionally, the referrer fee calculation applies inconsistent rounding methods. The origination fee uses rounding to the nearest integer (`to_round()`), while the referral fee always rounds down (`to_floor()`), creating bias in fee distribution.

Location: `calculate_borrow_all_available - Reserve::borrow`

```
// In calculate_borrow_all_available
let borrow_amount_f = /* fractional, e.g. 100.7 */
let borrow_amount: u64 = borrow_amount_f.to_floor(); // 100
let receive_amount = borrow_amount - origination_fee - referrer_fee;

// In Reserve::borrow
self.withdraw_freely_available(borrow_f.to_floor())?; // Withdraws 100
self.liquidity.borrowed_amount_sf = (borrowed_amount_f + borrow_f).to_bits(); //
Records 100.7
```

Location: `calculate_fees` function

```
let origination_fee: u64 = origination_fee_f.to_round(); // Rounds to nearest
```

```
let referral_fee = if need_to_assess_referral_fee {
    // ...
    let referral_fee_f = origination_fee_f * referral_fee_rate;
    referral_fee_f.to_floor::<u64>() // Always rounds down
}
```

The `add_to_withdrawal_accum` function is also affected as the `borrow_amount_f` is rounded up, increasing the value from 100.7 to 101.

Exploit scenario

1. Alice, a borrower, initiates a borrow operation for a fractional amount of 100.7 tokens. The system calculates the actual withdrawal as 100 tokens (rounded down) but records 100.7 tokens as borrowed debt in Alice's account. Alice receives 100 tokens but is charged interest on 100.7 tokens, creating phantom debt of 0.7 tokens.
2. Bob, a referrer, participates in the fee distribution system. When Alice borrows, the origination fee is calculated using the `to_round` method while Bob's referral fee uses the `to_floor` method. This inconsistent rounding creates systematic bias where referral fees are consistently lower than they should be, reducing Bob's expected compensation.

Recommendation

Implement consistent rounding methods across all borrow calculations:

- use the same rounding approach for both debt recording and actual token withdrawal amounts;
- apply uniform rounding to all fee calculations to eliminate bias in fee distribution; and
- ensure that recorded debt matches the actual withdrawn amount.

Go back to Findings Summary

W1: Insufficient token account fixability validation in withdrawal queue

Impact:	Warning	Likelihood:	N/A
Target:	-	Type:	Logic error

Description

The `withdraw_queued_liquidity` instruction executes the `resolve` method whose source code can be seen below.

Listing 14. Excerpt from handler_withdraw_queued_liquidity

```
138 pub fn resolve(
139     account: &AccountInfo,
140     owner: &Pubkey,
141     mint: &Pubkey,
142     token_program: &Pubkey,
143 ) -> Self {
144     if account.lamports() == 0 {
145         // account does not exist...
146         let ata_address =
147             get_associated_token_address_with_program_id(owner, mint,
    token_program);
148         return if account.key() == ata_address {
149             Self::Fixable // ...but can be created permissionlessly
150         } else {
151             Self::NonFixable // ...and can only be created by the owner
152         };
153     }
154     let Ok(token_account) = TokenAccount::try_deserialize(&mut
    account.data.borrow().as_ref())
155     else {
156         return Self::NonFixable; // not a token account
157     };
158     if &token_account.owner != owner || &token_account.mint != mint {
159         return Self::NonFixable; // the owner and mint must match the
    expected (stored) values
160     }
161     if token_account.delegate.is_some() {
162         return Self::NonFixable; // the "delegate" feature must not be used
    at all (just repeating the constraint enforced on enqueue - see the
```

```
    rationale in the comment there)
163     }
164     Self::Valid
165 }
```

The issue with this method is that the Associated Token Account is marked as fixable only in case it has zero lamports. If intentionally or mistakenly someone transfers as few as 1 lamport into an already closed associated token account, it would prevent the protocol from marking the account as fixable.

Exploit scenario

Alice is a regular user.

Bob is a user monitoring a Solana block explorer.

1. Alice queues collateral for withdrawal.
2. Alice mistakenly closes her token account meant for liquidity withdrawal.
3. Bob notices that the token account was closed while it is meant for liquidity withdrawal.
4. Bob sends 1 lamport to the closed token account.
5. The protocol marks the token account as non-fixable.
6. Alice needs to recover her collateral.

Recommendation

Allow closed associated token accounts with non-zero lamports to be considered fixable.

Fix 1.1

The issue was fixed by removing the zero lamport check and verifying the account owner and the data field.

Go back to Findings Summary

W2: Token account validation bypasses program owner checks

Impact:	Warning	Likelihood:	N/A
Target:	-	Type:	Data validation

Description

The `withdraw_queued_liquidity` instruction executes the `resolve` method whose source code can be seen below.

Listing 15. Excerpt from handler_withdraw_queued_liquidity

```
138 pub fn resolve(
139     account: &AccountInfo,
140     owner: &Pubkey,
141     mint: &Pubkey,
142     token_program: &Pubkey,
143 ) -> Self {
144     if account.lamports() == 0 {
145         // account does not exist...
146         let ata_address =
147             get_associated_token_address_with_program_id(owner, mint,
    token_program);
148         return if account.key() == ata_address {
149             Self::Fixable // ...but can be created permissionlessly
150         } else {
151             Self::NonFixable // ...and can only be created by the owner
152         };
153     }
154     let Ok(token_account) = TokenAccount::try_deserialize(&mut
    account.data.borrow().as_ref())
155     else {
156         return Self::NonFixable; // not a token account
157     };
158     if &token_account.owner != owner || &token_account.mint != mint {
159         return Self::NonFixable; // the owner and mint must match the
    expected (stored) values
160     }
161     if token_account.delegate.is_some() {
162         return Self::NonFixable; // the "delegate" feature must not be used
    at all (just repeating the constraint enforced on enqueue - see the
```

```
    rationale in the comment there)
163     }
164     Self::Valid
165 }
```

The issue with this function is on the line that deserializes the token account. The `try_deserialize` method does not enforce any account ownership validation, meaning any data account (with valid data length) can be deserialized into a token account. This can lead to account spoofing.

In this particular case, the issue is marked only as a warning as it is important for the client to be aware of it. The issue is not exploitable due to the token transfer happening later in the instruction, where the transaction will revert if an incorrect data account is sent in place of a token account.

Recommendation

Check the program owner of the token account before deserialization.

Fix 1.1

The issue was fixed by checking that the account is either `system_program` owned (Fixable) or `token_program` owned (Valid).

Go back to Findings Summary

W3: Withdrawal queue system lacks cancellation mechanism enabling administrative DoS

Impact:	Warning	Likelihood:	N/A
Target:	-	Type:	Logic error

Description

The withdrawal queue system contains design flaws that enable denial-of-service attacks through administrative controls combined with the complete absence of a user cancellation mechanism. Users who create withdrawal tickets cannot cancel them, making them dependent on administrative permissions to access their funds.

The system has two administrative controls that can be weaponized or used mistakenly:

Redemption Control: The `withdraw_ticket_redemption_enabled` flag can disable all ticket redemptions.

Listing 16. Excerpt from handler_withdraw_queued_liquidity

```
require!(
    lending_market.is_withdraw_ticket_redemption_enabled(),
    LendingError::WithdrawTicketRedemptionDisabled,
);
```

Minimum Value Threshold: The `min_withdraw_queued_liquidity_value` setting can block small withdrawals.

Listing 17. Excerpt from lending_operations

```
fn check_min_withdrawn_queued_liquidity_value(
    lending_market: &LendingMarket,
    reserve: &Reserve,
    liquidity_amount_to_withdraw: u64,
    ticket_emptied: bool,
```

```
) -> Result<()> {
    if ticket_emptied {
        return Ok(()); // if this closes a ticket, we have to allow any amount (otherwise it would be a liveness-break)
    }
    let liquidity_value_to_withdraw = calculate_market_value_from_liquidity_amount(
        reserve,
        Fraction::from_num(liquidity_amount_to_withdraw),
    );
    if liquidity_value_to_withdraw < lending_market.min_withdraw_queued_liquidity_value {
        xmsg!(
            "Withdrawn liquidity {} would have value {}, lower than the configured minimum {}",
            liquidity_amount_to_withdraw,
            liquidity_value_to_withdraw.to_display(),
            lending_market.min_withdraw_queued_liquidity_value
        );
        return err!(LendingError::WithdrawQueuedLiquidityValueTooSmall);
    }
    Ok(())
}
```

If there is no option for cancellation and these flags are used on purpose or mistakenly, users will have their assets locked until manual intervention by the responsible authority.

Recommendation

Allow users to inactivate their tickets. This means tickets will still be present in the queue to be able to call `withdraw_queued_liquidity`. However, if the ticket is inactive, collateral is withdrawn by the user and the ticket can be dequeued.

Go back to Findings Summary

W4: Missing Minimum Notice Period for Debt Maturity Allows Instant Liquidation

Impact:	Warning	Likelihood:	N/A
Target:	-	Type:	Code quality

Description

The `debt_maturity_timestamp` validation only prevents past timestamps but allows setting maturity to the current timestamp, giving users zero time to repay loans before liquidation becomes possible.

Combined with an unbounded `deleveraging_threshold_decrease_bps_per_day` parameter that could reach millions of basis points per day, positions can become eligible for liquidation faster than users expect.

While not as severe as retroactive timestamp setting, this configuration presents several issues:

- no minimum notice period exists (should be at least hours);
- breaks user expectations since users cannot execute transactions before liquidation eligibility; and
- enables front-running potential where administrators could sandwich user transactions.

While this behavior could be intentional by design, further investigation is needed to determine the intended functionality.

Recommendation

Implement a minimum notice period for debt maturity timestamp changes (example 12h). Require that `debt_maturity_timestamp` be set to at least several hours or days in the future to provide users adequate time to repay loans or

take defensive actions before liquidation becomes possible.

Go back to Findings Summary

W5: Lack of validation in LendingMarket name allows duplicates/impersonation confusion

Impact:	Warning	Likelihood:	N/A
Target:	-	Type:	Data validation

Description

There is no validation for the name field within the `LendingMarket` struct.

The UX feature ”Advanced Search” allows users to query `LendingMarket` by its name field.

`BorrowLendList.js` Advanced search allows market listing by name (any name):

` return e.jsx(ce, { withSearch: !0, withSelectAll: !0, onChange: a ⇒ o(a.map(i ⇒ i.marketAddress)), selectedItems: r, items: t, searchLabel: ”Search markets…”, searchKey: ”label”, itemKey: ”marketAddress”,`

`BorrowLendList.js` label field from market is populated by `getName()`:

` , Ft = () ⇒ { const {markets: t} = A() , {activeReserves: o} = Ie() , {isAssetDeprecated: s} = Ce() , r = g.useMemo( () ⇒ t.map(n ⇒ ({ marketAddress: n.address, label: n.getName(),`

`vendor-kamino-all` `getName()` queries the label directly from the name field of the on-chain `LendingMarket` account, decoding it from UTF-8 and likely stripping null bytes (`\0`) if padded.

Recommendation

Implement validation for the `LendingMarket` name field to ensure uniqueness and prevent impersonation. Consider maintaining a registry of used names or implementing a naming convention that includes unique identifiers.

Go back to Findings Summary

I1: Collateral vault persists after complete withdrawal recovery

Impact:	Info	Likelihood:	N/A
Target:	-	Type:	Logic error

Description

The `recover_invalid_ticket_collateral` instruction allows users to recover collateral from invalid tickets. However, the instruction does not close the `owner_queued_collateral_vault` when it becomes empty.

Recommendation

Allow the `owner_queued_collateral_vault` to be closed when it becomes empty after recovery.

Go back to Findings Summary

I2: UserMetadata: Unused and Unvalidated PDA field

Impact:	Info	Likelihood:	N/A
Target:	-	Type:	Code quality

Description

`user_lookup_table` does not seem to be used anywhere in the code, is populated directly from user input, and is not validated.

Recommendation

Remove the unused `user_lookup_table` field from the PDA structure or implement proper validation if it serves a purpose.

Go back to Findings Summary

Report Revision 1.1

Revision Team

Member's Name	Position
Andrej Lukačovič	Lead Auditor
Matej Hyčko	Auditor
Max Kupchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 1.1.

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Audit Report | Kamino: Kamino Lending, 18.5.2026.

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz