

Kamino Klend

Security Assessment

March 30th, 2026 — Prepared by OtterSec

Tamta Topuria	tamta@osec.io
Artem Astakhov	aikasimirov@osec.io
Ajay Shankar Kunapareddy	d1r3wolf@osec.io
Xiang Yin	soreatu@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **6**

Vulnerabilities **7**

OS-KLD-ADV-00 | Context-Dependent Zero Rollover Semantics 9

OS-KLD-ADV-01 | Missing Elevation Limit Checks 11

OS-KLD-ADV-02 | Lack of Mint Validation and Utilization of Shared Vaults 12

OS-KLD-ADV-03 | Elevation Group Rollover Bypass 14

OS-KLD-ADV-04 | Withdraw Queue Tombstone Spam 15

OS-KLD-ADV-05 | Denial of Service on Borrow Capacity via Partial Rollover 16

General Findings **18**

OS-KLD-SUG-00 | Misleading Ownership Transfer Helper Name 19

OS-KLD-SUG-01 | Inaccurate Ownership Transfer Error Semantics 20

OS-KLD-SUG-02 | Overly Strict Rollover Configuration Equality 21

OS-KLD-SUG-03 | Redundant Code 23

Appendices

Vulnerability Rating Scale **25**

Procedure **26**

© 2026 Otter Audits LLC. All Rights Reserved.

01 — Executive Summary

Overview

Exponent Finance engaged OtterSec to assess the `klend` PRs. This assessment was conducted between February 15th and March 27th, 2025.

For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 10 finding throughout this audit engagement.

In particular, we identified a high-risk vulnerability where zero-valued rollover parameters are interpreted differently on fixed-term versus open-term reserves, but are propagated unchanged across rollovers, enabling an attacker to utilize a fixed → open → fixed rollover path to place the user onto a fixed-term reserve with a much higher rate or much shorter term than the user intended (OS-KLD-ADV-00). Additionally, same-reserve rollover ignores elevation-specific borrow caps, allowing debt above newly lowered contextual limits to be renewed indefinitely, bypassing governance's attempt to wind down borrowing in that specific context (OS-KLD-ADV-01). Also, same-reserve rollover is not blocked for elevation-group obligations, even though that path skips the elevation-specific checks and accounting updates those positions require (OS-KLD-ADV-03).

Furthermore, canceled withdraw tickets remain as zero-amount tombstones in the FIFO queue, so an attacker can cheaply spam and cancel tickets to force others to spend transactions clearing them. This degrades withdrawal liveness and raises keeper/user processing costs (OS-KLD-ADV-04).

We also suggested modifying certain helper functions to utilize proper naming conventions (OS-KLD-SUG-00) and to improve error-handling logic (OS-KLD-SUG-01). We further advised removing unutilized (OS-KLD-SUG-02), and redundant code instances (OS-KLD-SUG-03).

© 2026 Otter Audits LLC. All Rights Reserved.

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Kamino-Finance/klend. Audits were performed on the following PRs:

1. PR#632
2. PR#633
3. PR#642
4. PR#640
5. PR#646
6. PR#711
7. PR#720
8. PR#728
9. PR#742
10. PR#700
11. PR#668
12. PR#736
13. PR#704
14. PR#677
15. PR#708
16. PR#743
17. PR#714
18. PR#657

© 2026 Otter Audits LLC. All Rights Reserved.

A brief description of the program is as follows:

© 2026 Otter Audits LLC. All Rights Reserved.

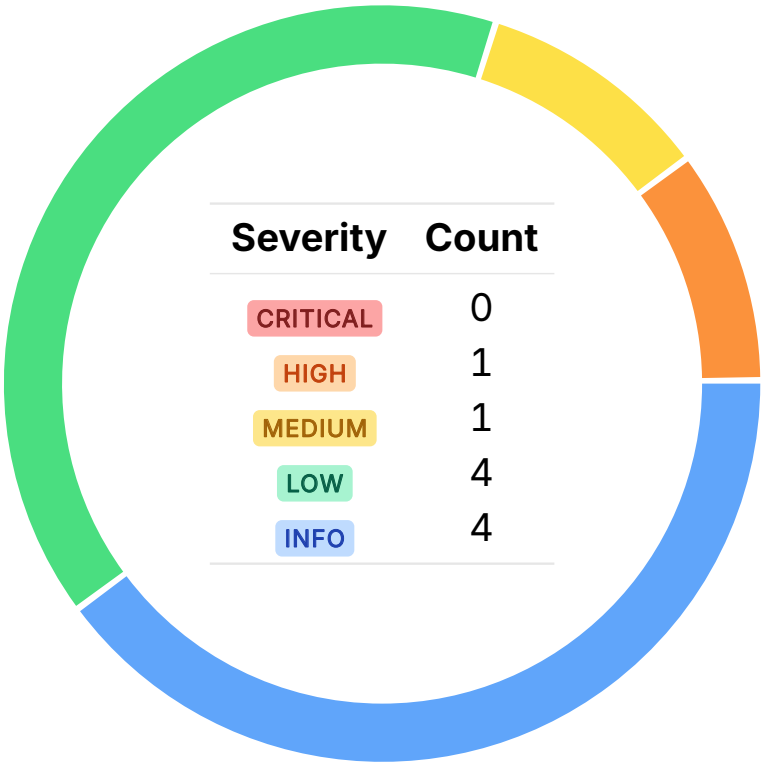
Name	Description
Rollover-PR's	Enables fixed-term borrowers opt into permissionless loan renewal shortly before maturity, avoiding manual repay-and-reborrow cycles. The borrower still controls the outcome through constraints such as maximum borrow rate, minimum term, and whether migration into open-term debt is allowed.
PR#700	Rate-limits (throttles) how much of an expired fixed-term debt can be liquidated over time.
PR#668	Clones one reserve's config onto another (same-mint, pre-deposit) reserve, with optional overrides.
PR#736	Fixes freshness checks and applies stricter minimum-value constraints for `BO`s and `WT`s.
PR#704	Adds `fraction_collateral_to_liquidity_ceil` to `CollateralExchangeRate`.
PR#677	Implements a secure three-step process for transferring obligation ownership between users.
PR#708	Cancels withdraw ticket handler implementation via tombstoning.
PR#743	Implements Early Repay Penalty v2 by introducing a remaining-interest-based penalty calculation for fixed-term borrows, integrating the penalty into the repayment flow, and updating the configuration and tests to support the new model.
PR#714	Disallows aborting an obligation ownership transfer once it has been approved by the global admin.
PR#657	Improves the precision of `approximate_compounded_interest` and fixes the affected tests.

© 2026 Otter Audits LLC. All Rights Reserved.

03 — Findings

Overall, we reported 10 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	1
LOW	4
INFO	4

© 2026 Otter Audits LLC. All Rights Reserved.

04 — Vulnerabilities

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

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-KLD-ADV-00	HIGH	RESOLVED	Zero-valued rollover parameters such as `max_borrow_rate_bps = 0` and `min_debt_term_seconds = 0` are interpreted differently on fixed-term versus open-term reserves, but are propagated unchanged across rollovers, enabling an attacker to utilize a `fixed → open → fixed` rollover path to place the user onto a fixed-term reserve with a much higher rate or much shorter term than the user intended.
OS-KLD-ADV-01	MEDIUM	RESOLVED	Same-reserve rollover ignores elevation-specific borrow caps, allowing debt above newly lowered contextual limits to be renewed indefinitely, bypassing governance's attempt to wind down borrowing in that specific context.
OS-KLD-ADV-02	LOW	RESOLVED	The rollover utilization check uses the fractional `repaid_amount` instead of the ceiling-rounded `borrowed_amount` actually added to the target reserve, potentially underestimating the utilization increase by one token unit and allowing rollover past the configured utilization limit.
OS-KLD-ADV-03	LOW	RESOLVED	Same-reserve rollover is not blocked for elevation-group obligations, even though that path skips the elevation-specific checks and accounting updates those positions require.

© 2026 Otter Audits LLC. All Rights Reserved.

OS-KLD-ADV-04	LOW	RESOLVED	Canceled withdraw tickets remain as zero-amount tombstones in the FIFO queue, so an attacker can cheaply spam and cancel tickets to force others to spend transactions clearing them. This degrades withdrawal liveness and raises keeper/user processing costs.
OS-KLD-ADV-05	LOW	RESOLVED	Partial rollovers may split one borrow across multiple low-liquidity reserves, filling the obligation's limited `borrows` array, blocking the user from opening future borrows, creating a denial-of-service on borrowing capacity.

© 2026 Otter Audits LLC. All Rights Reserved.

Context-Dependent Zero Rollover Semantics HIGH OS-KLD-ADV-00

Description

In the current implementation, `max_borrow_rate_bps == 0` and `min_debt_term_seconds == 0` are interpreted contextually, not uniformly. When `max_borrow_rate == 0`, it has two different meanings depending on whether the target is a fixed-term reserve or an open-term reserve. In a fixed-term context, a user may reasonably understand zero as preserving the current reserve's fixed-term rate behavior, but in an open-term context, the same stored value is interpreted as "no maximum rate restriction." Since, in certain rollover paths, the configuration is propagated into the new borrow configuration, a user may, in some cases, be rolled into a reserve with a borrow rate materially higher than the rate specified in the original configuration. The Same logic applies to `min_debt_term_seconds`.

Considering a `fixed -> open -> fixed` sequence:

1. A user has a fixed-term borrow with `auto_rollover_enabled = 1`, `open_term_allowed = 1`, `migration_to_fixed_enabled = 1`, `max_borrow_rate_bps = 0`.
2. An attacker creates a transaction with the two rollover instructions.
    - Rollover into an open-term reserve. With this instruction, the configuration is propagated to the new borrow. In this open-term context, `max_borrow_rate = 0` is interpreted as allowing any borrow rate, meaning the rollover may proceed without enforcing an upper rate bound.
    - Rollover into a fixed-term reserve. Attacker's logic selects the reserve with the highest `max_borrow_rate`.
3. Consequently, the borrow will be rolled into a fixed-term reserve with an arbitrarily high allowed rate, even if the borrower set the value to zero expecting the borrow rate to remain unchanged from the original reserve.

>_ *klend/src/lending_market/lending_operations.rs* RUST

```rust
/// Determines whether the [ObligationLiquidity::fixed_term_borrow_rollover_config] should be
    ↪ left
/// as-is (rather than being cleared) after the given rollover operation.
fn preserves_rollover_config(source_borrow: &ObligationLiquidity, target_reserve: &Reserve,) ->
    ↪ bool {
    // If the target reserve is fixed-term, then we want to keep rolling over the same way in
        ↪ the future...
    target_reserve.config.get_debt_term_seconds().is_some()
        // ... and even if migrating to open-term, we want to allow the migration trip back *if
            ↪ the user opted-in* for that:
        || source_borrow.fixed_term_borrow_rollover_config.is_migration_to_fixed_enabled()
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

Additionally, `min_debt_term_seconds = 0` is initially utilized as a sentinel, implying that `BorrowOrder` may only be filled by an open-term reserve. However, once `fill_borrow_order` propagates that value into `fixed_term_borrow_rollover_config` together with `migration_to_fixed_enabled = 1`, the same raw `min_debt_term_seconds` is later reutilized in a fixed-term rollover context. In that later context, zero no longer enforces open-term only and instead behaves as an effectively unconstrained minimum debt term.

As a result, a borrow that was validly filled on an open-term reserve may later be rolled into a fixed-term reserve with a very short term. This changes the effective borrower protections without any explicit revalidation of the original intent.

Remediation

1. Remove the dual interpretation of zero values for `max_borrow_rate` and `min_debt_term_seconds` in the `ObligationLiquidity` configuration.
2. Retain the interpretation of `min_debt_term_seconds = 0` as indicating that the borrow may only be filled by, or migrated to, an open-term reserve across `BorrowOrder` and rollover parameters.

Patch

Resolved in PR#749.

© 2026 Otter Audits LLC. All Rights Reserved.

Missing Elevation Limit Checks MEDIUM OS-KLD-ADV-01

Description

The rollover path currently validates only the reserve's global `borrow_limit` and `utilization_limit`, both with `borrow_amount = 0`, to ensure the reserve is not already in a disallowed state. However, it does not enforce the elevation-group-specific limits that apply to the obligation's current context: `borrow_limit_outside_elevation_group` for non-`eMode` debt and `borrow_limit_against_this_collateral_in_elevation_group[i]` for elevation-group borrowing against a particular collateral bucket. Thus, if the admin lowers these limits below the amounts already borrowed in order to wind down a specific borrowing context, same-reserve rollover would still allow those loans to be extended indefinitely, undermining the admin's intended restriction.

>_ klend/src/lending_market/lending_operations.rs RUST

```rust
pub fn rollover_borrow_into_same_reserve(
    market: &LendingMarket,
    reserve_address: Pubkey,
    reserve: &Reserve,
    obligation: &mut Obligation,
    clock: &Clock,
) -> Result<()> {
    // Check all the preconditions for rollover:
    let (borrow, borrow_index) = obligation.find_liquidity_in_borrows(reserve_address)?;
    check_rollover_possible([...])?;
    check_same_reserve_rollover_liquidity_available(reserve, borrow)?;
    check_below_reserve_borrow_limit(reserve, Fraction::ZERO)?;
    check_below_reserve_utilization_limit(reserve, Fraction::ZERO)?;

    // The actual "rollover" operation looks disappointingly easy in the "same reserve" case:
    let borrow = obligation.get_active_borrow_mut(borrow_index)?; // re-borrow the borrow, this
        ↪ time mut
    borrow.last_borrowed_at_timestamp = u64::try_from(clock.unix_timestamp).unwrap(); //
        ↪ normally, any borrow does this; here we have to move the timestamp manually, because
        ↪ we do not borrow even 1 lamport more
    [..]
}
```

Remediation

Add a check in same-reserve rollovers to ensure that the borrow complies with the `borrow_limit_outside_elevation_group` limit.

Patch

Resolved in PR#751.

© 2026 Otter Audits LLC. All Rights Reserved.

Lack of Mint Validation and Utilization of Shared Vaults LOW OS-KLD-ADV-02

Description

In `lending_operations::rollover_borrow_into_different_reserve`, the target-side debt is ultimately opened utilizing `borrowed_amount`, which is defined as the ceiling-rounded whole-token amount actually borrowed from the target reserve. However, the utilization pre-check is performed with `repaid_amount` (as `check_below_reserve_utilization_limit` is called with `repaid_amount`), which is the pre-rounding fractional source debt amount. This creates a mismatch between the amount utilized for validation and the amount that is actually applied to reserve accounting.

>_ *klend/src/lending_market/lending_operations.rs* RUST

```rust
pub fn rollover_borrow_into_different_reserve([...]) -> Result<FixedTermRolloverResult> {
    [...]
    // Rollover into a different reserve means "repaying in the source reserve accounting,
        ↪ borrowing
    // in the target reserve accounting, and transferring the actual funds between the
        ↪ reserves":
    let repaid_amount = source_borrow.borrowed_amount(); // we have to repay all source (no
        ↪ "partial rollover" supported)
    let tokens_to_transfer_over = repaid_amount.to_ceil::<u64>(); // when we transfer, we need
        ↪ full lamports, though
    let borrowed_amount = Fraction::from(tokens_to_transfer_over); // which means we actually
        ↪ have to borrow this much from the target (to maintain its "total supply" invariant)
    let obligation_borrowed_amount_increase = borrowed_amount - repaid_amount; // so the
        ↪ rollover's user's debt may increase by a fraction of lamport
    [...]
    // Make sure the moved borrow would not violate the target reserve's configured limits:
    check_below_reserve_borrow_limit(target_reserve, Fraction::from(borrowed_amount))?;
    check_below_reserve_utilization_limit(target_reserve, repaid_amount)?;
    [...]
}
```

Because `borrowed_amount = ceil(repaid_amount)`, the validated amount is always either equal to, or exactly one token unit smaller than, the debt actually added to the target reserve. If the reserve is close to its utilization threshold, this 1-unit gap will result in `check_below_reserve_utilization_limit` to pass even though the subsequent rollover pushes utilization above the configured limit.

The adjacent absolute borrow-limit check already utilizes `borrowed_amount`, and the regular borrow flow likewise validates utilization utilizing the actual borrowed amount rather than the pre-rounding debt figure. The rollover path is therefore inconsistent with the protocol's normal borrow semantics.

© 2026 Otter Audits LLC. All Rights Reserved.

Remediation

Ensure `check_below_reserve_utilization_limit` is called with `repaid_amount` instead of `borrowed_amount`.

Patch

Resolved in PR#646.

© 2026 Otter Audits LLC. All Rights Reserved.

Elevation Group Rollover Bypass LOW OS-KLD-ADV-03

Description

Same-reserve rollover currently remains available even when the obligation belongs to an elevation group, despite that path not performing the elevation-specific accounting and validation that other borrowing flows rely on. This allows constrained elevation-group positions to extend fixed-term debt through a simplified path that was not designed for them.

Remediation

Block same-reserve rollovers for obligations with an elevation group.

Patch

Resolved in PR#751.

© 2026 Otter Audits LLC. All Rights Reserved.

Withdraw Queue Tombstone Spam LOW OS-KLD-ADV-04

Description

Fully canceled withdraw tickets remain in the FIFO queue as zero-amount tombstones until their sequence number is processed by `withdraw_queued_liquidity`. This allows an attacker to repeatedly create minimum-sized tickets and cancel them immediately, filling the queue with tombstone entries that still must be traversed in order. Because the ticket rent is eventually refunded, the attacker's effective cost is mostly limited to transaction fees and temporary capital lockup. This turns cancellation into a griefing scenario as honest users or keepers must spend additional transactions processing `withdraw_queued_liquidity` repeatedly just to skip tombstones and advance the queue.

>_ *klend/src/handlers/handler_cancel_withdraw_ticket.rs* RUST

```rust
/// Process the cancel withdraw ticket instruction.
///
/// Allows a ticket owner to cancel their queued withdrawal (partially or fully) and receive
    ↪ ctokens
/// back immediately.
///
/// Even a fully-cancelled ticket is NOT closed - it remains in the queue as a "tombstone" (==
    ↪ 0).
/// The rent is only returned when the ticket's turn is reached in the queue (i.e. processed by
/// `withdraw_queued_liquidity`). This discourages potential "enqueue + cancel" spam by imposing
    ↪ a
/// holding cost on cancelled tickets.
pub fn process([...])
```

Remediation

Charge an explicit cancellation fee sufficient to compensate the payer or keeper responsible for clearing cancelled tickets. As an operational fallback, ticket cancellation may also be disabled through configuration if spam is observed.

Patch

This issue was acknowledged by Kamino.

© 2026 Otter Audits LLC. All Rights Reserved.

Denial of Service on Borrow Capacity via Partial Rollover LOW OS-KLD-ADV-05

Description

`lending_operations::rollover_borrow_into_different_reserve` allows partial rollovers whenever the target reserve cannot absorb the full source debt, as long as the migrated slice exceeds `market.min_partial_rollover_value`. That threshold only constrains the economic size of each partial migration; it does not constrain the number of borrow slots the rollover sequence may consume in the obligation.

>_ *klend/src/lending_market/lending_operations.rs* RUST

```rust
pub fn rollover_borrow_into_different_reserve(
    market: &LendingMarket,
    source_reserve_address: Pubkey,
    source_reserve: &mut Reserve,
    target_reserve_address: Pubkey,
    target_reserve: &mut Reserve,
    obligation: &mut Obligation,
    clock: &Clock,
) -> Result<FixedTermRolloverResult> {
    [...]
    // For partial rollovers, enforce the minimum dollar value threshold:
    let is_full_rollover = Fraction::from(tokens_to_transfer_over) >= full_borrow_amount;
    if !is_full_rollover {
        let rollover_value = calculate_market_value_from_liquidity_amount(
            target_reserve,
            Fraction::from(tokens_to_transfer_over),
        );
        if rollover_value < market.min_partial_rollover_value {
            msg!(
                "Partial rollover value {} below minimum {}",
                rollover_value.to_display(),
                market.min_partial_rollover_value
            );
            return err!(LendingError::PartialRolloverValueTooSmall);
        }
    }
    [...]
}
```

A permissionless rollover caller may therefore split one borrow across several low-liquidity reserves that each satisfy the threshold, causing the obligation's `borrows` array to fill up. Once the array is exhausted, the user may be unable to open borrows from additional reserves even though their economic position remains otherwise healthy, resulting in a denial-of-service on future borrowing capacity.

© 2026 Otter Audits LLC. All Rights Reserved.

Remediation

Ensure that partial rollover cannot create an excessive number of borrow positions.

Patch

This issue was acknowledged by Kamino.

© 2026 Otter Audits LLC. All Rights Reserved.

05 — General Findings

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

ID	Description
OS-KLD-SUG-00	The ownership transfer helper name is misleading because it suggests the cleanup only happens during ownership-transfer initiation, while it is actually reutilized across multiple transfer phases.
OS-KLD-SUG-01	The helper checks for any ownership-transfer-in-progress state, but its error and log message refer specifically to the Initiated phase. This makes the semantics misleading and may confuse future callers about what condition actually failed.
OS-KLD-SUG-02	Borrow merging currently relies on exact configuration equality instead of behavioral equivalence, so semantically identical rollover preferences can still be treated as incompatible. This may unnecessarily block merges and fragment borrows, affecting user experience.
OS-KLD-SUG-03	The codebase contains multiple cases of redundant code that should be removed for better maintainability and clarity.

© 2026 Otter Audits LLC. All Rights Reserved.

Misleading Ownership Transfer Helper Name OS-KLD-SUG-00

Description

`clear_expired_borrow_order_on_initiating_obligation_ownership_transfer` in `lending_operations` suggests that the function is only meant to run during the initiation phase of an obligation ownership transfer, while it is also during approval and acceptance. Thus, the phase-specific name does not accurately describe its actual utilization scope and may mislead readers about the helper's actual role in the ownership-transfer flow.

>_ *klend/src/lending_market/lending_operations.rs* RUST

```rust
pub fn clear_expired_borrow_order_on_initiating_obligation_ownership_transfer(
    obligation: &mut Obligation,
    clock: &Clock,
) -> Result<()> {
    let timestamp = clock.unix_timestamp.try_into().expect("negative timestamp");

    obligation
        .borrow_order
        .clear_if_past_fillable_timestamp(timestamp);

    Ok(())
}
```

Remediation

Utilize a phase-agnostic name such as `clear_expired_borrow_order_before_obligation_ownership_transfer` to clarify that the helper performs generic ownership-transfer preparation regardless of the specific transfer stage.

Patch

This issue was acknowledged by Kamino.

© 2026 Otter Audits LLC. All Rights Reserved.

Inaccurate Ownership Transfer Error Semantics OS-KLD-SUG-01

Description

`obligation::check_ownership_transfer_in_progress` validates only whether the obligation is in any active ownership-transfer state, but its failure path refers specifically to the `Initiated` phase, as observed by the `ObligationOwnershipTransferNotInInitiatedState` error and the corresponding error message. This is semantically inaccurate and may confuse future callers. Also, the function appears to be unutilized.

>_ *klend/src/state/obligation.rs* RUST

```rust
pub fn check_ownership_transfer_in_progress(&self) -> Result<()> {
    if !self.is_ownership_transfer_in_progress() {
        xmsg!("Obligation ownership transfer not initiated");
        return err!(LendingError::ObligationOwnershipTransferNotInInitiatedState);
    }
    Ok(())
}
```

Remediation

Rename the message and error to something that clearly describes it as a failure of a generic in-progress check, or remove the helper if unutilized.

Patch

This issue was acknowledged by Kamino.

© 2026 Otter Audits LLC. All Rights Reserved.

Overly Strict Rollover Configuration Equality OS-KLD-SUG-02

Description

In `lending_operations`, both `check_rollover_into_existing_slot_possible` and `propagate_rollover_config_to_borrow` currently require structural equality of `FixedTermBorrowRolloverConfig`, implying every field must match byte-for-byte. However, it is possible for two different configurations to map to the same effective rollover policy in the current context. For example, if a configuration permits only `Fixed -> Open` rollover, then fields such as `max_borrow_rate_bps` or `target_term` may be semantically irrelevant to that path.

>_ *klend/src/lending_market/lending_operations.rs* RUST

```rust
pub fn rollover_borrow_into_different_reserve([...]) -> Result<FixedTermRolloverResult> {
    [...]
    // Rollover into a different reserve means "repaying in the source reserve accounting,
        ↪ borrowing
    // in the target reserve accounting, and transferring the actual funds between the
        ↪ reserves":
    let repaid_amount = source_borrow.borrowed_amount(); // we have to repay all source (no
        ↪ "partial rollover" supported)
    let tokens_to_transfer_over = repaid_amount.to_ceil::<u64>(); // when we transfer, we need
        ↪ full lamports, though
    let borrowed_amount = Fraction::from(tokens_to_transfer_over); // which means we actually
        ↪ have to borrow this much from the target (to maintain its "total supply" invariant)
    let obligation_borrowed_amount_increase = borrowed_amount - repaid_amount; // so the
        ↪ rollover's user's debt may increase by a fraction of lamport
    [...]
    // Make sure the moved borrow would not violate the target reserve's configured limits:
    check_below_reserve_borrow_limit(target_reserve, Fraction::from(borrowed_amount))?;
    check_below_reserve_utilization_limit(target_reserve, repaid_amount)?;
    [...]
}
```

Thus, a configuration with `auto: true, open: true, migration: false, rate: 500, term: 0` produces the same result as one with `auto: true, open: true, migration: false, rate: 1, term: 0`. Both effectively permit only `Fixed -> Open` rollover, where `max_borrow_rate` configuration is ignored. Two borrows that would roll over in exactly the same way are therefore treated as incompatible solely because of unused parameter differences. Another example where different configurations produce the same rollover behavior include: `auto: false, open: true, migration: true, rate: N, term: K` and `auto: false, open: false, migration: true, rate: N, term: K`

© 2026 Otter Audits LLC. All Rights Reserved.

As a result, in `fill_borrow_order`, two borrows with effectively identical behavior will not be merged if their configurations differ. In `propagate_rollover_config_to_borrow`, this implies an order cannot fill into a pre-existing borrow slot whose rollover semantics are functionally identical but structurally different. In `check_rollover_into_existing_slot_possible`, it may prevent source and target borrows from merging during rollover even though their effective rollover permissions are the same. Thus, while this does not result in a security issue, users may end up with unnecessary fragmentation or unexpected fill failures even though there is no meaningful behavioral conflict.

Remediation

Implement a custom comparison operator for `FixedTermBorrowRolloverConfig`, or canonicalize `RolloverConfig` so semantically equivalent configurations are normalized to the same representation. For example, when `auto = false` and `migration = true`, force `rate = 0` and `term = 0`, ensuring that configurations with identical behavior compare equal through normalization at configuration time rather than during rollover.

Patch

This issue was acknowledged by Kamino.

© 2026 Otter Audits LLC. All Rights Reserved.

Redundant Code

OS-KLD-SUG-03

Description

1. In `lending_operations::rollover_borrow_into_different_reserve`, `borrowed_amount` is already a `Fraction`, so calling `check_below_reserve_borrow_limit` with `Fraction::from(borrowed_amount)` is redundant and results in unnecessary conversion. Remove the redundant conversion.

>_ klend/src/lending_market/lending_operations.rs RUST

```rust
pub fn rollover_borrow_into_different_reserve(
    market: &LendingMarket,
    source_reserve_address: Pubkey,
    source_reserve: &mut Reserve,
    target_reserve_address: Pubkey,
    target_reserve: &mut Reserve,
    obligation: &mut Obligation,
    clock: &Clock,
) -> Result<FixedTermRolloverResult> {
    [...]
    let borrowed_amount = Fraction::from(tokens_to_transfer_over);
    [...]
    // Make sure the moved borrow would not violate the target reserve's configured
        ↪ limits:
    check_below_reserve_borrow_limit(target_reserve, Fraction::from(borrowed_amount))?;
    [...]
}
```

2. `ReserveConfig::max_borrow_rate_bps` scans the entire `borrow_rate_curve.points` array and computes the maximum dynamically, even though the curve has a fixed structure and the maximum rate is expected to be represented by the last point. Replace the iteration with direct indexing, returning `borrow_rate_curve.points[10].borrow_rate_bps`.

>_ klend/src/state/reserve.rs RUST

```rust
/// Returns the maximum rate achievable on the [Self::borrow_rate_curve], in bps.
pub fn max_borrow_rate_bps(&self) -> u32 {
    self.borrow_rate_curve
        .points
        .iter()
        .map(|point| point.borrow_rate_bps)
        .max()
        .expect("curve has a static >0 length")
}
```

© 2026 Otter Audits LLC. All Rights Reserved.

Remediation

Remove the redundant code instances highlighted above.

Patch

1. Resolved in PR#646.
2. This issue was acknowledged by Kamino.

© 2026 Otter Audits LLC. All Rights Reserved.

A — Vulnerability Rating Scale

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

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

© 2026 Otter Audits LLC. All Rights Reserved.

B — Procedure

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

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

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

© 2026 Otter Audits LLC. All Rights Reserved.