certora

Security Assessment

Final Report



Throttle term-based liquidations & Klend Auto-rollover feature

March 2026

Prepared for Kamino

certora

Table of Contents

Project Summary **3**
- Project Scope 3
- Project Overview 5
- Protocol Overview 5

Assessment Methodology **6**

Threat and Security Overview **7**
- **Findings Summary** **8**
- Severity Matrix 8

Detailed Findings **9**
- **High Severity Issues** **11**
  - H-01 Different-Reserve Rollover Merges Borrows Without Handling Existing State 11
- **Medium Severity Issues** **14**
  - M-01 Enabling Referral Fees Would Break “refresh_obligation” for All Referred Obligations That Rolled Into a Fresh Reserve 14
  - M-02 Same reserve rollover doesn't check for reserve limits 16
  - M-03 Mint isn't checked for rollover, allowing a transfer with fees at the expense of the protocol 17
  - M-04 Rollover Configuration Lock-In When Market Disables Feature 18
  - M-05 “borrowed_amount_at_expiration” Not Reset on Rollover Leads to Under-Protected Throttle Window 19
- **Low Severity Issues** **21**
  - L-01 “min_partial_rollover_value” isn't checked to not be zero when enabling migration to fixed 21
  - L-02 Reserve Wind-Down via `debt_maturity_timestamp` Does Not Trigger the Rollover Window 22
  - L-03 Implementation Contradiction in Rollover Configuration Bounds 24
  - L-04 Skipped rollover config check when the market level rollover is disabled 26
  - L-05 “borrowed_amount_at_expiration” can be inflated, gaming the system 27
- **Informational Severity Issues** **29**
  - I-01: Different-Reserve Rollover Does Not Validate Borrow Factor Parity 29

Disclaimer **31**

About Certora **31**

certora

Project Summary

Project Scope

Auto-rollover related PRs:

Project Name	PR Links	Commit Hash	Platform
On-chain state changes + specs for fixed-term auto-rollover - PR632	https://github.com/Kamino-Finance/klend-private/pull/632	11a9d176d3e30d155960699808124e147731cc20	Solana
Allow configuring auto-rollover opt-ins on Obligation - PR633	https://github.com/Kamino-Finance/klend-private/pull/633	e68bef9a56baf8b7149672e36a5fcafb3eb46bf7	Solana
Implement auto-rollover handler for same reserve case - PR640	https://github.com/Kamino-Finance/klend-private/pull/640	94ab20072bb70a46b9ee57b3b9b0cc0f482ea8ab	Solana
Propagate auto-rollover settings from borrow order to its filled borrows - PR642	https://github.com/Kamino-Finance/klend-private/pull/642	5fa63eeddf1a11ae6ff160e887802b53ea1e18d7	Solana
Rollover handler logic for the "different reserve" case - PR646	https://github.com/Kamino-Finance/klend-private/pull/646	3235d89d405c2c7c4eded4b57757295f3d40e5e5	Solana
Support migration from open-term to fixed-term in rollover - PR711	https://github.com/Kamino-Finance/klend-private/pull/711	b6ae79495a2db7724f0d4687bf095e47a2c669ae	Solana

Support partial rollover to different reserve - PR720	https://github.com/Kamino-Finance/klend-private/pull/720	d72de52729ca55402e2ccb1a516e8c1a5c0e0b28	Solana
Rollover into existing borrow only when extra conditions met - PR728	https://github.com/Kamino-Finance/klend-private/pull/728	bb67c2f6a96b451ae57b9e308901b606567cf5bc	Solana
Treat Rollover's max_borrow_rate (> 0) and min_debt_term_seconds (0 == open-term only) same as in BOs - PR749	https://github.com/Kamino-Finance/klend-private/pull/749	7d52b8d4ae9d6f83e49bfe506ea49c6ac51c0dec	Solana
Add check for "borrow limit outside elevation group" in Rollover to same reserve - PR751	https://github.com/Kamino-Finance/klend-private/pull/751	7d52b8d4ae9d6f83e49bfe506ea49c6ac51c0dec	Solana

Throttle term-based liquidations related PRs:

Throttle term-based liquidations	https://github.com/Kamino-Finance/klend-private/pull/700	c7efc29cce78df2a73882cc972423e51ccc8ac64	Solana

Project Overview

This document describes the security review of **Klend Auto-rollover feature under different PRs**. The work was undertaken from **February 9th, 2026** to **February 13th, 2026** and from March 1**3th, 2026 to March 17th, 2026**.

The scope includes all changes made in **programs/klend/src** from the PRs mentioned above

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

Protocol Overview

Kamino Lending (KLend) is a decentralized finance (DeFi) platform designed to optimize the borrowing and lending of crypto assets on the Solana blockchain. The protocol team actively develops new features and updates the already deployed program.

KLend supports **fixed-term borrows** with an **expiration timestamp**. When a fixed-term borrow expires, it becomes eligible for liquidation. The **auto-rollover** feature lets users opt in to having their borrows automatically extended before expiration, preventing this.

There are three rollover types: **fixed-to-fixed** (roll into another fixed-term reserve), **fixed-to-open** (fall back to an open-term reserve), and **open-to-fixed** (migrate an open-term borrow into a fixed-term reserve). Users configure rollover per-borrow via FixedTermBorrowRolloverConfig, setting a maximum acceptable borrow rate, minimum debt term, and which flavors are permitted.

Execution is permissionless — any caller (e.g. a bot) can trigger it within market-defined time windows before expiration. In the **same-reserve** case, only accounting is updated (the clock is reset, no transfers). In the **different-reserve** case, liquidity is borrowed from the target reserve and transferred to repay the source reserve. No origination fees are charged on rollover, and the rollover config is propagated to the new borrow slot after execution.

A throttle term-based liquidations feature is triggered when a fixed-term borrow expires (ObligationBorrowDebtTermReached). Instead of allowing the full position to be liquidated immediately, the protocol throttles how much can be liquidated based on how long ago the term ended.

Specifically, at expiry the borrow's size is snapshotted as **borrowed_amount_at_expiration**. Over the next **term_based_full_liquidation_duration_secs** (market-level config), the liquidatable portion ramps linearly from 0% to 100%.

Assessment Methodology

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

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

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

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

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

certora

Threat and Security Overview

Actors and Trust Model

- **Protocol Admin** — Trusted. Enables/disables rollover execution and configuration globally.
- **Obligation Owner** — Trusted for own state. Opts in/out per borrow; sets rollover config bounds.
- **Permissionless Executor** — Untrusted. Triggers rollover on any opted-in obligation; chooses source and target reserve accounts.
- **Liquidator** — Untrusted. Liquidates expired borrows; interacts with the throttle mechanism.

The core trust tension: the owner configures rollover constraints, but the executor chooses which reserve to roll into and when. Any path where the executor can override or corrupt the owner's intent is a critical failure.

Assets at Risk

- **Borrow position integrity**: borrow slot amounts and rollover configurations
- **Liquidation throttle protection:** borrowed_amount_at_expiration, governing how fast an expired position can be liquidated
- **Reserve liquidity**: token balances in source and target reserve vaults
- **Obligation operability**: the ability to refresh, borrow, repay, and liquidate

Entry Points

- rollover_fixed_term_borrow — Core handler; same-reserve or different-reserve path.
- update_obligation_config — Owner updates per-borrow rollover flags and bounds.
- update_lending_market — Admin enables/disables rollover windows and configuration flag.
- fill_borrow_order — Creates borrow slots and propagates rollover config.

- refresh_obligation — Must remain operable for all obligations post-rollover.

Threat Categories

Permissionless Execution Abuse

Threat: Permissionless actor corrupts rollover config via borrow merge — A different-reserve rollover that lands on a reserve where the obligation already has a borrow silently merges amounts and overwrites the existing FixedTermBorrowRolloverConfig with the source borrow's config. The executor can widen, restrict, or disable the user's protective bounds without their consent.

Maps to: H-01 (Fixed)

Threat: Reserve capacity limits bypassed on same-reserve rollover — The same-reserve path skips utilization and capacity checks. Accrued interest can push a reserve over its limits between borrows, allowing a rollover to succeed on a reserve that should be considered full.

Maps to: M-02 (Fixed)

Accounting Integrity

Threat: borrowed_amount_at_expiration not reset on rollover — first_borrowed_at_timestamp is reset to start a new term but borrowed_amount_at_expiration is not cleared. When the new term expires the throttle formula uses a stale snapshot, exposing the borrower to faster liquidation than the protocol guarantees.

Maps to: M-05 (Fixed)

Threat: borrowed_amount_at_expiration inflation gaming — A borrower takes an extra borrow just before expiry, triggers refresh_obligation post-expiry to lock in an inflated snapshot, then partially repays. The throttle formula produces near-zero max liquidatable amount for most of the ramp, blocking liquidation far longer than intended.

Maps to: L-05 (Fixed)

Token Transfer Safety

Threat: Missing mint validation enables fee-on-transfer tokens to drain the protocol — The different-reserve rollover transfer does not run the same mint checks as borrow_obligation_liquidity. A fee-on-transfer token causes the source vault to receive less than the amount recorded in accounting; the fee comes at the protocol's expense.

Maps to: M-03 (Fixed)

Threat: Borrow factor disparity between source and target reserves — No check at rollover time validates that borrow_factor_pct matches between reserves. Rolling from a lower to a higher factor can immediately push the obligation into a liquidatable state on the next refresh.

Maps to: I-01 (Fixed)

Configuration Integrity

Threat: User cannot opt out when market disables rollovers with both flags set — The configuration guard fires after each single-field update. With both auto_rollover_enabled and migration_to_fixed_enabled on and the market flag disabled, neither can be disabled individually without triggering the guard, trapping the user in an opted-in state.

Maps to: M-04 (Fixed)

Threat: min_partial_rollover_value not validated for open-to-fixed migration — The non-zero check applied when enabling fixed-to-fixed or fixed-to-open rollover is absent when enabling obligation_borrow_migration_to_fixed_execution_enabled.

Maps to: L-01 (Fixed)

Threat: Config mismatch check skipped when market disables rollovers — propagate_rollover_config_to_borrow silently skips the mismatch check when rollover is

market-disabled, allowing a borrow order with conflicting rollover intent to fill and leave the user with unintended rollover behavior.

Maps to: L-04 (Fixed)

Threat: max_borrow_rate_bps spec implies equal-or-better guarantees the code does not enforce — No validation at configuration or execution time prevents setting a value that allows worse terms than the current reserve. The documentation was misleading about the protective semantics.

Maps to: L-03 (Fixed — documentation updated)

Feature Interactions and Latent Issues

Threat: Referral fee activation would break refresh_obligation for rolled obligations — Different-reserve rollover does not initialize the ReferrerTokenState PDA for (obligation.referrer, target_reserve). Currently latent since referral fees are zero on all deployed markets. If fees are enabled, refresh_obligation fails for every affected obligation — blocking liquidation — until each PDA is manually initialized. The backlog grows silently with each rollover.

Maps to: M-01 (Acknowledged — referral feature likely to be deprecated)

Reserve wind-down via debt_maturity_timestamp bypasses the rollover window — The rollover window is computed solely from the per-borrow deadline and is unaware of debt_maturity_timestamp. An admin winding down a reserve can make borrows liquidatable via the maturity path before their rollover window ever opens, negating the protection auto-rollover is meant to provide.

Maps to: L-02 (Acknowledged)

Invariants

Config sovereignty: the executor must never be able to alter the owner's rollover configuration.

Reserve limits hold post-rollover: both reserves must satisfy utilization and capacity limits after execution.

No net liquidity creation: borrowed amount must not increase beyond interest-accrued pre-rollover debt.

Vault balance consistency: token transfers must exactly match accounting updates in both reserves.

Throttle snapshot correctness: borrowed_amount_at_expiration must reflect actual debt at the start of each new expiry period.

Obligation operability: refresh_obligation must remain executable for any obligation after a rollover.

Opt-out always possible: a user must always be able to disable all rollover flags regardless of market configuration state.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	-	-	-
High	1	1	1
Medium	5	5	4
Low	5	5	4
Informational	1	1	1
Total	12	12	10

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
H-01	Different-Reserve Rollover Merges Borrows Without Handling Existing State	High	Fixed
M-01	Enabling Referral Fees Would Break “refresh_obligation” for All Referred Obligations That Rolled Into a Fresh Reserve	Medium	Acknowledged
M-02	Same reserve rollover doesn't check for reserve limits	Medium	Fixed
M-03	Mint isn't checked for rollover, allowing a transfer with fees at the expense of the protocol	Medium	Fixed
M-04	Rollover Configuration Lock-In When Market Disables Feature	Medium	Fixed
M-05	“borrowed_amount_at_expiration” Not Reset on Rollover Leads to Under-Protected Throttle Window	Medium	Fixed
L-01	“min_partial_rollover_value” isn't checked to not be zero when enabling migration to fixed	Low	Fixed
L-02	Reserve Wind-Down via “debt_maturity_timestamp” Does Not Trigger the Rollover Window	Low	Acknowledged

L-03	Implementation Contradiction in Rollover Configuration Bounds	Low	Fixed
L-04	Skipped rollover config check when the market level rollover is disabled	Low	Fixed
L-05	“borrowed_amount_at_expiration” can be inflated, gaming the system	Low	Fixed
I-01	Different-Reserve Rollover Does Not Validate Borrow Factor Parity	Informational	Fixed

High Severity Issues

H-01 Different-Reserve Rollover Merges Borrows Without Handling Existing State

Severity: High	Impact: High	Likelihood: Medium
Files: rollover_borrow_into_different_reserve() in lending_market/lending_operations.rs (PR646, lines 993-1003)	Status: Fixed	Related to PR646

Description:

When a different-reserve rollover targets a reserve where the obligation already has an active borrow, the rolled-over amount is silently merged into the existing borrow. This merge is not accounted for in the rollover logic, producing two concrete issues.

The mechanism: The rollover flow proceeds as follows (lines 993-1003 of lending_operations.rs):

1. `obligation.repay_fully(repaid_amount, source_borrow_index)` — zeroes out the source borrow slot entirely (line 994, calls `obligation.rs:241` which sets the slot to `ObligationLiquidity::default()`)
2. `obligation.find_or_add_liquidity_to_borrows(target_reserve_address, ...)` — looks up borrows by reserve address (line 995-998). If an existing borrow on the target reserve is found (`obligation.rs:397-398`), it returns that borrow **unchanged** — the `cumulative_borrow_rate` and `current_timestamp` parameters are ignored
3. `target_borrow.borrow(borrowed_amount)` — adds the rolled amount to the existing borrow's `borrowed_amount_sf` (line 1000, calls `obligation.rs:724-726`)
4. `target_borrow.fixed_term_borrow_rollover_config = preserved_rollover_config` — unconditionally overwrites the existing borrow's rollover config with the source borrow's config (line 1003)

Example scenario:

A user has an obligation with two USDC borrows:

- Borrow A on Reserve X: 10,000 USDC, max_borrow_rate_bps = 2000, min_debt_term_seconds = 86400, auto-rollover enabled
- Borrow B on Reserve Y: 50,000 USDC, max_borrow_rate_bps = 500, min_debt_term_seconds = 2592000 (30 days), auto-rollover enabled

A bot rolls Borrow A (Reserve X) into Reserve Y. The result:

- Borrow B on Reserve Y now holds 60,000 USDC
- The rollover config is overwritten: max_borrow_rate_bps = 2000, min_debt_term_seconds = 86400 (from Borrow A)
- The user's carefully chosen protective bounds on their 50,000 USDC position are silently replaced
- The combined 60,000 USDC can now be rolled into a reserve with up to 2000 bps rate and as little as 1-day term

Impact:

A permissionless actor can alter the rollover configuration protecting a user's existing borrow by triggering a different-reserve rollover that merges into it. This can widen the bounds (allowing worse rollover terms than the user intended), restrict the bounds (blocking rollovers the user wanted), or disable auto-rollover entirely. The user has no on-chain mechanism to prevent this once auto-rollover is enabled on both borrows.

Recommendations:

The rollover_borrow_into_different_reserve() function should check whether the target reserve already has an active borrow on the obligation. If it does, either:

1. **Reject the rollover**: Return an error if the obligation already has a borrow on the target reserve. This is the simplest and safest approach.
2. **Handle the merge explicitly**: If merging is desired, the config should be merged deliberately (e.g., taking the more restrictive of each field: `min(max_borrow_rate_bps)`, `max(min_debt_term_seconds)`, rollover enabled only if

both had it enabled). The timestamp should also be reset to the current time, consistent with the same-reserve rollover path.

Customer Response:

Resolved in PR728 If there is an existing borrow in the target Reserve, the rollover config must be identical or the rollover will be aborted. Additionally the existing borrow end-date must be greater than the source borrow.

Fix Review:

The fix resolves the issue. One edge-case is that rollover from fixed to open will almost always be aborted if the client has not explicitly set the borrow in open term to be the same as the fixed term. This was acknowledged in a comment by the protocol.

Medium Severity Issues

M-01 Enabling Referral Fees Would Break “refresh_obligation” for All Referred Obligations That Rolled Into a Fresh Reserve

Severity: Medium	Impact: High	Likelihood: Low
Files: lending_market/lending_operations.rs (PR646)	Status: Acknowledged	Related to PR646

Description:

borrow_obligation_liquidity enforces that the ReferrerTokenState PDA for (obligation.referrer, borrow_reserve) is initialized before creating a borrow slot — returning ReferrerAccountMissing if absent (handler_borrow_obligation_liquidity.rs:96). rollover_borrow_into_different_reserve() creates a borrow slot in the target reserve without this check, leaving the corresponding ReferrerTokenState PDA for (obligation.referrer, target_reserve) uninitialized when the target is fresh.

The market config files in configs/deployed/mainnet/ show that all six deployed markets (main, jito, jlp, altcoins, ethena, susd) have referral_fee_bps: 0. Since accumulate_referrer_fees only processes the ReferrerTokenState accounts when both obligation_has_referrer is true and referral_fee_bps > 0, the iterator is never consumed and the uninitialized account is never validated. So currently the issue is latent.

If referral_fee_bps is set above zero on any market, the iterator is consumed and refresh_obligation begins validating those accounts via FatAccountLoader::try_from(...).unwrap(). Every referred obligation that previously rolled into a fresh reserve would have an uninitialized PDA in the required account slot, causing refresh_obligation to fail. Since refresh_obligation is a prerequisite for all eight core handlers (enforced via check_refresh_ixs`) — including liquidation — those obligations become inoperable until init_referrer_token_state is separately called for each affected (obligation.referrer, target_reserve) pair.

The scope of the disruption grows over time. Each different-reserve rollover on a referred obligation adds one more uninitialized PDA to the backlog silently and without any on-chain indication. The longer referral_fee_bps remains at zero, the larger the backlog. When referral fees are eventually enabled every obligation in that backlog breaks simultaneously. Recovery is one init_referrer_token_state transaction per affected (referrer, reserve) pair. The larger the protocol grows before fees are enabled, the more significant the manual intervention required and the longer liquidation remains unavailable on a potentially large set of positions.

Recommendations:

When obligation.has_referrer() and the target reserve differs from the source, require the ReferrerTokenState PDA for (obligation.referrer, target_reserve) in RolloverAccounts using init_if_needed semantics — consistent with setup.rs:364–371.

Customer Response:

Acknowledged - “this feature is not enabled anywhere and we will probably deprecate it fully“

M-02 Same reserve rollover doesn't check for reserve limits

Severity: Medium	Impact:	Likelihood:
Files: programs/klend/src/lending_market/lending_operations.rs	Status: Fixed	Related to PR640

Description:

While executing the same reserve rollover we don't check that the reserve limits didn't exceed.

The assumption is that since the borrowed amount in the reserve didn't increase then this check isn't needed. However, it's possible that the limits have been passed due to interest. And in that case we shouldn't allow the rollover to happen.

Recommendations:

Check for reserve capacity and utilization limits for same-reserve rollover as well

Customer Response:

Fixed in 944eb7f

Fix Review:

We found a minor issue with the fix, where the repaid_amount was used instead of the borrowed amount. This was later fixed in commit 2b75790.

Other than that, no issues were found with the fix.

M-03 Mint isn't checked for rollover, allowing a transfer with fees at the expense of the protocol

Severity: Medium	Impact: Medium	Likelihood: Medium
Files: programs/klend/src/lending_market/lending_checks.rs	Status: Fixed	Related to PR640

Description:

While executing the rollover we don't check the mint like we do with other operations like borrowing, even though we transfer funds when the rollover is between different reserves.

This would allow for example to do a rollover while fee transfer is non-zero, and the fee would be at the expense of the protocol (we'd assume the target reserve received the full amount, while the fees were deducted from it)

Recommendations:

Run those checks for rollover

Customer Response:

Fixed in commit f12a24c.

Fix Review:

Fix confirmed.

M-04 Rollover Configuration Lock-In When Market Disables Feature

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: programs/klend/src/lending_market/lending_operations.rs - check_can_configure_auto_rollover	Status: Fixed	Related to PR749

Description:

The check_can_configure_auto_rollover guard in lending_operations.rs is applied after each individual field update. Since UpdateObligationConfigMode only allows changing one field per instruction, a user with both auto_rollover_enabled=true and migration_to_fixed_enabled=true cannot disable either flag individually — disabling one still leaves the other enabled, which triggers the guard.

The code's own comment states the intent is to *always* allow opt-out:
"Note: a user is always allowed to opt-out from the auto-rollover"

Recommendations: Always allow user to out-out.

Customer's response: Fixed in [be888e34e67d60187d6bc7464432eee709bda144]

Fix Review: Fix confirmed.

M-05 “borrowed_amount_at_expiration” Not Reset on Rollover Leads to Under-Protected Throttle Window

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: programs/klend/src/state/obligation.rs:749-764	Status: Fixed	Related to PR700

Description:

borrowed_amount_at_expiration is a one-shot snapshot: capture_borrowed_amount_at_expiration captures the current debt the first time the borrow is found to be expired, then returns early on all subsequent calls:

```rust
// obligation.rs:754
if self.borrowed_amount_at_expiration != 0 {
    return; // already captured
}
```

The rollover handler resets first_borrowed_at_timestamp to start a new term but does not clear borrowed_amount_at_expiration:

```rust
// lending_operations.rs:929
borrow.first_borrowed_at_timestamp = u64::try_from(clock.unix_timestamp).unwrap();
// borrowed_amount_at_expiration is not touched
```

When the new term expires, capture_borrowed_amount_at_expiration skips because the field is non-zero, so the throttle formula:

```rust
// liquidation_operations.rs:54-58

throttle_protected = borrowed_amount_at_expiration × (secs_until_full_liq / full_duration)

max_liquidatable  = borrowed_amount - throttle_protected
```

uses the old (lower) snapshot instead of the actual debt at the new expiry. Since interest accrued over the new term, borrowed_amount_at_expiration < actual_debt_at_new_expiry, so throttle_protected is smaller than intended and the borrower is exposed to faster liquidation than the protocol guarantees.

Recommendations: Update borrowed_amount_at_expiration on rollover

Customer's response: Fixed.

Fix Review: Fix confirmed.

Low Severity Issues

L-01 “min_partial_rollover_value” isn't checked to not be zero when enabling migration to fixed

Severity: Low	Impact: Low	Likelihood: Medium
Files: programs/klend/src/handlers/handler_update_lending_market.rs	Status: Fixed	Related to PR720

Description:

In PR #720 (partial rollover support) we ensure that min_partial_rollover_value isn’t zero when enabling fixed-to-fixed or fixed-to-open term rollovers. However, this check isn’t present when enabling open-to-fixed term (via obligation_borrow_migration_to_fixed_execution_enabled).

Recommendations:

Add the necessary check

Customer Response:

Fixed in commit 6e62073.

Fix Review:

Fix looks good.

L-02 Reserve Wind-Down via `debt_maturity_timestamp` Does Not Trigger the Rollover Window

Severity: Low	Impact: Medium	Likelihood: Low
Files: check_rollover_possible() in lending_market/lending_operations.rs (PR646)	Status: Acknowledged	

Description:

When an admin winds down a reserve by setting debt_maturity_timestamp, auto-rollover should be the natural mechanism to migrate all fixed-term borrows to safety. In practice the two features are decoupled and work against each other.

The rollover window is computed exclusively from the per-borrow deadline (first_borrowed_at_timestamp + debt_term_seconds). It has no awareness of debt_maturity_timestamp. A borrow opened early in its 90-day term has its rollover window at day ~89. If the admin sets debt_maturity_timestamp = now + 20 days, that borrow becomes liquidatable on day 20 via the time-based path — at 100% of the debt with an escalating bonus — while the rollover window is still ~69 days away. Auto-rollover cannot intervene.

A reserve wind-down is precisely the scenario where the protocol most wants borrows migrated. Yet the auto-rollover system is blind to debt_maturity_timestamp and offers no protection in this case.

Recommendations:

Extend check_rollover_possible() to treat an imminent debt_maturity_timestamp on the source reserve as an alternative trigger for opening the rollover window, independently of where each borrow is in its per-borrow term

Customer Response:

Acknowledged. The debt_maturity_timestamp is from a feature which is concurrently being developed and the protocol at this time does not want both features to interact.

L-03 Implementation Contradiction in Rollover Configuration Bounds

Severity: Low	Impact: Low	Likelihood: Low
Files: Spec: `FixedTermBorrowRolloverConfig` struct documentation in `state/obligation.rs` (PR632, unchanged through PR646)	Status: Fixed	

Description:

The rustdoc specification on FixedTermBorrowRolloverConfig (introduced in PR632 and unchanged through PR646) describes the max_borrow_rate_bps field as a protective constraint that only allow equal-or-better terms:

> max_borrow_rate_bps - "when not zeroed, the rollover may use a reserve with a maximum borrow rate **equal or lower** than the given one."

The default behavior enforces exact match with the source reserve's parameters (conditions A and B in the spec). The non-zero values are described as "customization settings that can lift" these conditions, implying a relaxation that still guarantees equal-or-better outcomes.

However, the implementation contains no validation — at any point in the lifecycle (configuration in PR633, propagation in PR642, or execution in PR640/PR646) — that prevents the user from setting values that allow worse-than-current terms:

• max_borrow_rate_bps can be set to any u32 value, including values far exceeding the current reserve's max borrow rate

At configuration time (update_obligation_config in PR633), the field is not validated against the current reserve's parameters. The borrow_reserve account is passed in the handler but is only used to identify which borrow to configure — the reserve's config is never loaded or consulted.

At execution time (resolve_rollover_mode in PR640/PR646), the check is:

```rust
// When max_borrow_rate_bps is non-zero:

if target_borrow_rate_bps > max_borrow_rate_bps { revert }
```

These checks validate the target reserve against the user's configured bounds, but NOT against the source reserve's actual parameters.

Recommendations:

Either update the Natspec to clarify that max_borrow_rate_bps are user-chosen bounds with no inherent guarantee of equal-or-better terms. Users accept responsibility for the values they set.

Or add validation at configuration and execution time.

Customer Response:

The natspec has been updated to clarify the intent.

Fix Review:

Fix confirmed.

L-04 Skipped rollover config check when the market level rollover is disabled

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files:	Status: Fixed	Related to PR749

Description: When the market has disabled rollovers, the mismatch check inside propagate_rollover_config_to_borrow:

```
Rust
    if borrow.fixed_term_borrow_rollover_config != rollover_config {
        msg!("Borrow order's rollover config does not match the pre-existing borrow slot's config");
        return err!(LendingError::ObligationBorrowRolloverConfigMismatch);}
```

is silently skipped. A BO with conflicting rollover intent can fill borrows on reserves with existing rollover config — the BO "succeeds" but silently fails to apply its intended rollover config.

The impact is that the user gets a filled borrow with rollover behavior they didn't intend.

Recommendations: Throw if the market has disabled rollovers, but the BO expect such on fulfillment.

Customer's response: Fixed in 101a6cad0626cb109b76081139731e5f94c14856

Fix Review: Fix confirmed.

L-05 “borrowed_amount_at_expiration” can be inflated, gaming the system

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files:	Status: Fixed	Related to PR749

Description:

borrowed_amount_at_expiration is captured once — on the first RefreshObligation after a borrow's debt term expires — and is never updated again. The capture uses borrowed_amount_sf at that moment.

Because borrow_obligation_liquidity only blocks additional borrows from the same reserve after expiry, an attacker can borrow extra from the same reserve just before expiry, trigger RefreshObligation post-expiry to seal borrowed_amount_at_expiration at the inflated value, then repay the extra debt he just incurred as a partial repayment (which only touches borrowed_amount_sf, leaving borrowed_amount_at_expiration intact).

The throttle formula in max_liquidatable_borrowed_amount

```
Rust
protected = borrowed_amount_at_expiration × (secs_until_full_liq / full_duration)

max_liquidatable = borrowed_amount - protected
```

now produces max_liquidatable ≈ 0 for most of the ramp duration, because borrowed_amount_at_expiration is significantly larger than borrowed_amount. Every liquidation attempt returns `LiquidationTooSmall` until secs_until_full_liq reaches zero or some value that produces > 0. This protects the user from liquidation for a much longer period of time than intended, effectively gaming the system and the throttling mechanism.

Recommendations: Update borrowed_amount_at_expiration on subsequent calls, this defeats the attack. Note that the tradeoff is that honest borrowers who partially repay post-expiry would see their protected amount shrink.

Customer's response: Fixed

Fix Review: Fix confirmed.

Informational Severity Issues

I-01: Different-Reserve Rollover Does Not Validate Borrow Factor Parity

Files:

rollover_borrow_into_different_reserve() in lending_market/lending_operations.rs (PR646)

Description:

The different-reserve rollover performs no LTV-related check that accounts for a difference in borrow_factor_pct between the source and target reserves. borrow_factor_pct is a per-reserve parameter, and if two reserves for the same token within the same market were configured with different values, a rollover from a lower to a higher factor could immediately push the obligation into deep liquidation territory upon the next refresh — without any check at rollover time. This could be exploited by actors seeking to maximise the liquidation bonus.

We cannot identify a legitimate reason for same-token reserves within the same market to carry different borrow_factor_pct values, and no such configuration is currently known to exist. We flag this for the client's awareness in case there are deployment considerations we are not aware of.

Recommendations:

Check if the borrow_factor_pct of the reserves matches.

Customer Response:

Fixed.

Fix Review:

Fix confirmed.

Disclaimer

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

About Certora

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

Certora also provides services such as auditing, formal verification projects, and incident response.