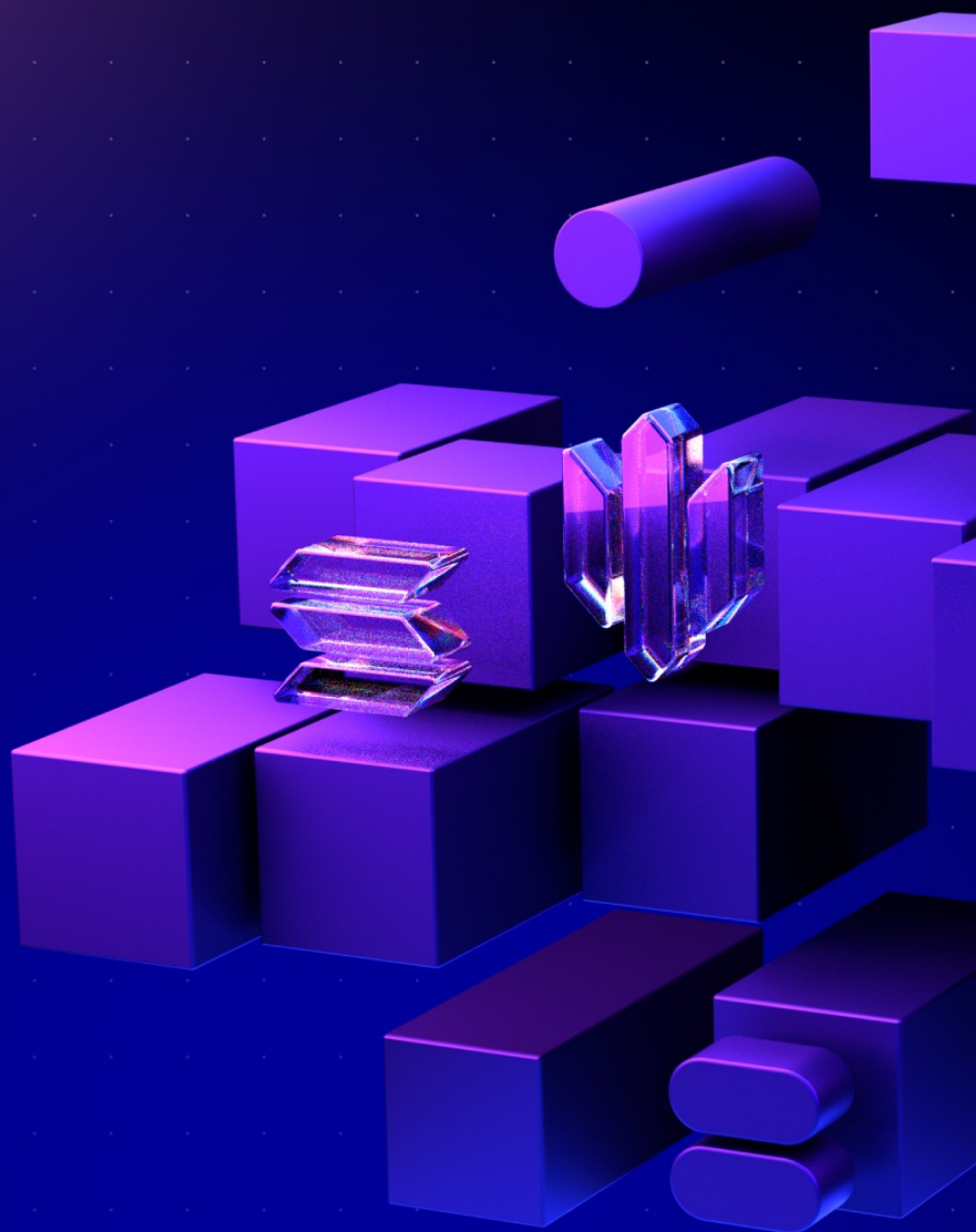
ackee
blockchain security

Kamino Lend

Fuzz tests

22.9.2025



Audit Report

Contents

1. Document Revisions

1.0-draft	Draft Report	29.01.2025
2.0-draft	Draft Report	11.08.2025
2.0	Final Report	14.08.2025
2.1	Fix Review	22.09.2025

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 fuzz testing methodology follows a systematic approach:

1. **Code and Architecture Analysis**
    a. High-level review of the Solana program specifications, Rust sources, and instruction handlers to understand the program's size, scope, and functionality.
    b. Analysis of Solana program entry points to identify instruction processors, account validation logic, and critical operations.
    c. Comparison of the Rust implementation and given specifications, ensuring that the program logic correctly implements everything intended.
2. **Fuzz Testing with Trident**
    a. **Interface Analysis**
        - Detailed examination of Solana instruction handlers and their account parameters
        - Identification of Program Derived Addresses (PDAs), account ownership, and cross-program invocation patterns
        - Mapping of account state transitions and Solana runtime data flows
    b. **Initial Behavior Exploration**
        - Writing simple Trident fuzz tests to observe Solana program instruction execution
        - Understanding account validation constraints and Solana runtime limitations
        - Identifying unexpected program behaviors, panics, or edge cases in

instruction processing

c. **Invariant Definition**

- Writing invariants based on expected Solana program properties and account state requirements
- Defining security-critical conditions for account ownership, balance constraints, and authority validation
- Establishing assertions for account state consistency and program-derived address integrity

d. **Complex Stateful Fuzz testing**

- Writing complex Trident fuzz tests that model stateful interactions across multiple Solana instructions
- Testing transaction sequences and their effects on account states and program data
- Exploring interdependencies between instruction handlers and cross-program invocations

e. **Extended Fuzz testing Campaigns**

- Running extended Trident fuzz testing campaigns to explore all edge cases in instruction execution
- Allowing the fuzzer to explore deep account state combinations and program execution paths
- Maximizing Rust code coverage and Solana instruction handler path exploration

f. **Dashboard Analysis**

- Continuous analysis of the Trident fuzz testing dashboard throughout the process

- Monitoring for program panics, instruction failures, and Rust code coverage metrics
- Identifying patterns that indicate potential Solana program vulnerabilities or runtime issues

3. **Vulnerability Assessment**
    a. Classification of discovered Solana program issues by severity and impact on protocol security
    b. Development of proof-of-concept transaction sequences for critical findings
    c. Recommendations for Rust code remediation based on Trident fuzz testing results

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
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 Lend is a decentralized lending platform deployed on the Solana blockchain that enables users to lend and borrow assets with flexible terms and interest rates.

Revision 1.0

Kamino engaged Ackee Blockchain Security to perform fuzz testing focused on the Kamino Lend protocol with a total time donation of 6 engineering days in a period between January 20 and January 30, 2025, with Andrej Lukačovič as the lead auditor. Manual code review was not performed.

The fuzz testing was performed on the commit `829c1f3`[1] and the scope was the following:

- Kamino Lending, excluding external dependencies.

We began our review by familiarizing ourselves with the protocol's interface and structure. This included understanding the instructions, accounts passed as instruction parameters, and inputs to the instructions. Additional initial steps were to familiarize ourselves with the protocol's main logic, including engaging with documentation and the frontend.

The next part was dedicated to writing simple fuzz tests to familiarize ourselves with instructions more deeply, to create a simple benchmark for which parts might be more difficult to fuzz, and to understand the whole flow of the scope. This included:

- writing fuzz tests for account initialization instructions;
- writing fuzz tests for operations on initialized accounts;
- writing fuzz tests for protocol state modifications; and

- writing fuzz tests for final execution paths.

After the initial part, we started to implement complex fuzz tests dedicated to the protocol's main logic. This included:

- creating independent fuzz tests for distinct protocol components;
- implementing invariant checks; and
- integrating protocol workspace configuration files with Trident.

During fuzz testing, we noticed that the following instructions were marked as deprecated, which were not included in the fuzz testing:

- `socialize_loss`
- `deposit_obligation_collateral`
- `withdraw_obligation_collateral`
- `borrow_obligation_liquidity`
- `repay_obligation_liquidity`
- `repay_and_withdraw_and_redeem`
- `deposit_reserve_liquidity_and_obligation_collateral`
- `withdraw_obligation_collateral_and_redeem_reserve_collateral`
- `liquidate_obligation_and_redeem_reserve_collateral`

During fuzz testing, we identified unhandled panics. Panicking behavior due to using edge cases of valid inputs can cause Denial of Service.

Our review resulted in 4 findings, ranging from Info to Warning severity.

Ackee Blockchain Security recommends Kamino:

- investigate the findings and severity of the issues;
- read and review the complete audit report; and

- address all identified issues.

See Report Revision 1.0 for the comprehensive fuzz testing description.

Revision 2.0

Kamino engaged Ackee Blockchain Security to perform fuzz testing focused on the Kamino Lend protocol with a total time donation of 15 engineering days in a period between June 23 and July 28, 2025, with Andrej Lukačovič as the lead auditor. Manual code review was not performed.

The fuzz testing was performed on the commit `fe1ad10`[2] and the scope was the following:

- Kamino Lending, excluding external dependencies.

The initial stage of the new fuzz testing revision included familiarizing ourselves with the changes made to the protocol since the last revision, as well as familiarizing ourselves with the protocol's interface.

In the next stage, we focused on the protocol's main logic, similarly to the previous revision:

- increasing the coverage of the fuzz tests;
- improving the number of executed transactions;
- investigating errors returned by instructions;
- writing invariant checks; and
- using stateful fuzz testing to guide the fuzzer specifically to the protocol's main logic based on the actual account states.

The following instruction were included in the fuzz testing:

- `init_lending_market`

- `update_lending_market`
- `update_lending_market_owner`
- `init_reserve`
- `redeem_fees`
- `withdraw_protocol_fee`
- `mark_obligation_for_deleveraging`
- `refresh_reserve`
- `deposit_reserve_liquidity`
- `redeem_reserve_collateral`
- `init_obligation`
- `refresh_obligation`
- `deposit_obligation_collateral_v2`
- `withdraw_obligation_collateral_v2`
- `borrow_obligation_liquidity_v2`
- `repay_obligation_liquidity_v2`
- `repay_and_withdraw_and_redeem`
- `deposit_and_withdraw`
- `deposit_reserve_liquidity_and_obligation_collateral_v2`
- `withdraw_obligation_collateral_and_redeem_reserve_collateral_v2`
- `liquidate_obligation_and_redeem_reserve_collateral_v2`
- `flash_repay_reserve_liquidity`
- `flash_borrow_reserve_liquidity`

Remaining instructions were not included in the fuzz testing because they were not required for testing the core logic of the protocol or they were

marked as deprecated.

During the fuzz testing multiple low severity issues were identified.

The W4 is caused by the fact that the borrow fee is not taken into consideration when validating the borrow amount.

The W5 is caused by the fact that the LTV to percentage conversion could cause overflow in scenarios where the price of the collateral drops significantly, which could lead to unexpected revert.

The W6 is caused by the fact that the owner is not marked as mutable, which could lead to unexpected revert.

The W7 is caused by the fact that the withdraw and repay instructions can unexpectedly revert due to panicking behavior.

Ackee Blockchain Security recommends Kamino:

- investigate the findings and severity of the issues;
- read and review the complete audit report; and
- address all identified issues.

See Report Revision 2.0 for the comprehensive fuzz testing description.

Revision 2.1

The review was done on the given commits `4c58439`[3], `89a6a81`[4], and `542ffdb`[5] respectively.

The findings reported in the previous revision were fixed, more precisely.

The W1 was fixed by adding a saturating arithmetic operation.

The W2 was fixed by adding a saturating arithmetic operation.

The W3 was acknowledged by the client. The instructions are executable only by the admin and the issue does not impose any direct security risks. Thus, the issue is reported only as warning.

The I1 was acknowledged by the client. The unused source code is present to avoid breaking changes.

The W4 was fixed by taking the borrow fee into consideration when validating the borrow amount.

The W5 was fixed by additing checked multiplication, preventing the overflow.

The W6 was fixed by marking the owner as mutable.

The W7 was fixed by not allowing withdraw and repay when the deposited value is zero.

[1] full commit hash: `829c1f3a701974a43a202f40b337106257106e2f`

[2] full commit hash: `fe1ad1056045d06d8dc8711b408e2384a6b836ee`

[3] full commit hash: `4c58439e7bbc9dde7fdeb47218ecd3aaf92a0482`

[4] full commit hash: `89a6a81c93e3fa172ebc1f9924a9a2cae37f51c2`

[5] full commit hash: `542ffdb9688dd9b62514586fef68dfd361f2b8cb`

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	0	0	0	7	1	8

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
W1: WithdrawObligationCollateralV2 subtraction overflow	Warning	1.0	Fixed
W2: RepayAndWithdrawAndRedeemV2 subtraction overflow	Warning	1.0	Fixed
W3: Unhandled panics	Warning	1.0	Acknowledged
I1: Unused code	Info	1.0	Acknowledged
W4: Borrow limit excludes fees when validating borrow amount	Warning	2.0	Fixed

Finding title	Severity	Reported	Status
W5: Liquidation instruction causes panic due to unwrap on None value	Warning	2.0	Fixed
W6: Withdraw obligation collateral instruction reverts due to immutable owner	Warning	2.0	Fixed
W7: Instructions cause panic due to division by zero when deposited value is zero	Warning	2.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

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

System Overview

The Kamino Lending protocol is a decentralized lending platform on the Solana blockchain that enables users to lend and borrow assets with flexible terms and interest rates.

Fuzz testing

Fuzz testing is an automated testing technique that discovers bugs by providing invalid, unexpected, or random data as inputs to a Solana program. Fuzz testing helps identify edge cases and vulnerabilities that might be missed during manual review.

Gray box fuzz testing with coverage-guided fuzzers like Honggfuzz and AFL represents a middle ground between black box (no knowledge of internals) and white box (full knowledge) testing approaches. In this methodology, developers and auditors guide the fuzzer by specifying which Solana instruction sequences should be executed and what accounts should be used, while the underlying fuzz testing engines automatically generate and mutate random input data. The coverage-guided aspect means the fuzzer learns from each execution, prioritizing inputs that explore new Rust code paths to maximize test coverage.

For this audit revision, pseudo fork testing was utilized to create a testing

environment that closely mimics the Solana Mainnet. This technique involves dumping real Solana accounts and programs from the Mainnet (such as on-chain accounts storing asset prices) and loading them into the TridentSVM execution environment. This approach ensures that the fuzz tests interact with realistic account data and program dependencies. The pseudo designation indicates that while we replicate Mainnet state, Trident cannot yet automatically fetch all required accounts from the desired cluster, requiring manual cloning of necessary Solana accounts and programs.

Additionally, property-based fuzz testing was employed to verify the correctness and expected behavior of Solana program instructions. This approach involves writing invariant checks—mathematical properties that must always hold true—which are automatically validated after each instruction executes successfully. These invariants allow us to compare the actual program behavior against expected behavior, catching subtle logic errors and account state inconsistencies. The fuzz testing campaign simulated a comprehensive spectrum of Solana system instructions while enforcing strong assertions about the program's behavior.

The complete list of implemented execution flows and invariants is available in Appendix B.

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

W1: WithdrawObligationCollateralV2 subtraction overflow

Impact:	Warning	Likelihood:	N/A
Target:	lending_market/lending_operations.rs	Type:	Arithmetics

Description

Fuzz testing discovered that during the `WithdrawObligationCollateralV2` instruction invocation, in the `post_withdraw_obligation_invariants` function, the subtraction overflow is triggered. More precisely the overflow is triggered in the following computation.

Listing 1. Excerpt from `post_withdraw_obligation_invariants.rs`

```
let new_unhealthy_borrow_value =
    Fraction::from_bits(obligation.unhealthy_borrow_value_sf)
    - asset_mv * Fraction::from_percent(reserve_liquidation_threshold_pct);
```

Exploit scenario

The numbers that triggered the overflow are the following:

```
collateral_amount: 1801439850948198 // input parameter
unhealthy_borrow_value_sf: 115292150460684697600000000000
asset_mv: 4152391473376.8241
reserve_liquidation_threshold_pct = 0.75
```

Recommendation

Ensure the computation will not trigger overflow.

Fix 2.0

The client fixed the issue by adding a saturating arithmetic operation.

Go back to Findings Summary

W2: RepayAndWithdrawAndRedeemV2 subtraction overflow

Impact:	Warning	Likelihood:	N/A
Target:	lending_market/lending_operations.rs	Type:	Arithmetics

Description

Fuzz testing discovered that during the `RepayAndWithdrawAndRedeemV2` instruction invocation, in the `post_withdraw_obligation_invariants` function, the subtraction overflow is triggered. More precisely the overflow is triggered in the following computation.

Listing 2. Excerpt from `post_withdraw_obligation_invariants`

```
let new_unhealthy_borrow_value =
    Fraction::from_bits(obligation.unhealthy_borrow_value_sf)
    - asset_mv * Fraction::from_percent(reserve_liquidation_threshold_pct);
```

Exploit scenario

The numbers that triggered the overflow are the following:

```
withdraw_collateral_amount: 93458986495443  // input parameter
repay_amount: 587790291434  // input parameter
unhealthy_borrow_value_sf: 115292150460684697600000000000
asset_mv: 21542673126.2729
reserve_liquidation_threshold_pct = 0.75
```

Recommendation

Ensure the computation will not trigger overflow.

Fix 2.0

The client fixed the issue by adding a saturating arithmetic operation.

Go back to Findings Summary

W3: Unhandled panics

Impact:	Warning	Likelihood:	N/A
Target:	handler_update_lending_market.rs, lending_market/lending_operations.rs	Type:	Code quality

Description

Fuzz testing found unhandled panics in the `update_lending_market` and `update_reserve_config` functions.

Recommendation

- handle panicking behavior correctly by throwing errors instead of panicking;
- ensure panics occur only for inputs that are not expected to be used by the admin.

Acknowledgment 2.0

The client acknowledged the panicking behavior. These instructions are executable only by the admin and the issue does not impose any direct security risks.

Go back to Findings Summary

I1: Unused code

Impact:	Info	Likelihood:	N/A
Target:	handler_init_referrer_token_state.rs	Type:	Code quality

Description

The `lending_market` account in the `InitReferrerTokenState` context is not necessary since the reserve always contain some `lending_market` address.

Recommendation

Remove the unused `lending_market` account from the `InitReferrerTokenState` context.

Acknowledgment 2.0

The client acknowledged the issue. The unused source code is present to avoid breaking changes.

Go back to Findings Summary

Report Revision 2.0

Revision Team

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

System Overview

The system overview remains unchanged from the previous revision.

Fuzz testing

Fuzz testing is an automated testing technique that discovers bugs by providing invalid, unexpected, or random data as inputs to a Solana program. For this audit revision, we employed Manually Guided Fuzzing, an advanced approach that builds upon traditional coverage-guided fuzz testing.

Manually Guided Fuzzing differs from the gray box fuzzing used in Revision 1.0 by giving auditors direct control over the testing process. While traditional coverage-guided fuzzers like Honggfuzz and AFL automatically generate random inputs to maximize code coverage, Manually Guided Fuzzing allows specification of exact input ranges, Solana instruction sequences, and test scenarios based on deep understanding of the program's logic and potential vulnerabilities. The key advantage is the ability to focus testing efforts on high-risk areas and complex account state interactions that automated fuzzers might miss.

A core component of this approach is stateful fuzz testing, which maintains awareness of the Solana program's current account state throughout the testing process. Unlike stateless testing where each test runs in isolation,

stateful fuzz testing reads the current account data and uses this information to generate contextually relevant random inputs for subsequent Solana instructions.

For this audit revision, pseudo fork testing was utilized to create a testing environment that closely mimics the Solana Mainnet. This technique involves dumping real Solana accounts and programs from the Mainnet and loading them into the TridentSVM execution environment. This approach ensures that the fuzz tests interact with realistic account data and program dependencies. The pseudo designation indicates that while we replicate Mainnet state, Trident cannot yet automatically fetch all required accounts from the desired cluster, requiring manual cloning of necessary Solana accounts and programs.

Property-based fuzz testing was employed to verify the correctness and expected behavior of Solana program instructions. This approach involves writing invariant checks—mathematical properties that must always hold true—which are automatically validated after each transaction executes successfully. These invariants allow us to compare the actual program behavior against expected behavior, catching subtle logic errors and account state inconsistencies.

A flow-based testing approach was implemented to guide the fuzzer through realistic usage patterns. Rather than testing Solana instructions in isolation, this method executes random permutations of instruction sequences that mirror real-world protocol usage. Each sequence of flows is randomly generated, ensuring comprehensive coverage of possible user interactions.

Finally, regression testing capabilities were implemented as requested by the client. This feature ensures that the state of Solana accounts remains consistent across multiple fuzz testing sessions, which is particularly valuable when verifying that program updates do not introduce unintended behavioral changes. By comparing account states before and after modifications, it is

possible to refactor Rust code while maintaining functional equivalence.

The complete list of implemented flows and invariants is available in Appendix B.

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

W4: Borrow limit excludes fees when validating borrow amount

Impact:	Warning	Likelihood:	N/A
Target:	programs/kamino-lending/src/lending_market/lending_operations.rs	Type:	Logic error

Description

The issue was discovered running the `usdc-sol` fuzzing session.

Fuzz testing discovered that during the borrow obligation liquidity instruction, the fees are calculated on top of the requested borrow amount. Although the instruction ensures the new amount after borrow will not be greater than the borrow limit, the instruction does not take into consideration the fees that are added on top of the borrow amount and are also transferred from the reserve.

The instruction first checks if the new borrow amount is not greater than the borrow limit.

Listing 3. Excerpt from programs/kamino-lending/src/lending_market/lending_operations.rs

```
pub fn borrow_obligation_liquidity<'info, T>(
    lending_market: &LendingMarket,
    borrow_reserve: &mut Reserve,
    obligation: &mut Obligation,
    liquidity_amount: u64,
    clock: &Clock,
    borrow_reserve_pk: Pubkey,
    referrer_token_state: Option<RefMut<ReferrerTokenState>>,
    deposit_reserves_iter: impl Iterator<Item = T>,
) -> Result<CalculateBorrowResult>

    // ...
```

```
    let borrow_limit_f = Fraction::from(borrow_reserve.config.borrow_limit);
    let new_borrowed_amount_f = liquidity_amount_f +
reserve_liquidity_borrowed_f;
    if liquidity_amount != u64::MAX && new_borrowed_amount_f > borrow_limit_f {
        msg!(
            "Cannot borrow above the borrow limit. New total borrow: {} > limit:
{}",
            new_borrowed_amount_f.to_display(),
            borrow_reserve.config.borrow_limit
        );
        return err!(LendingError::BorrowLimitExceeded);
    }

    // ...

}
```

However, later in the instruction, the fees are added to the borrow amount.

Listing 4. Excerpt from programs/kamino-lending/src/state/reserve.rs

```
pub fn calculate_borrow(
    &self,
    amount_to_borrow: u64,
    max_borrow_factor_adjusted_debt_value: Fraction,
    remaining_reserve_borrow: Fraction,
    referral_fee_bps: u16,
    is_in_elevation_group: bool,
    has_referrer: bool,
) -> Result<CalculateBorrowResult> {

    // ...

    let mut borrow_amount_f = Fraction::from(receive_amount);

    borrow_amount_f += Fraction::from_num(borrow_fee + referrer_fee);

    // ...

}
```

Recommendation

Ensure the validation of the borrow limit includes fees when validating the borrow amount.

Fix 2.1

The issue was fixed by taking the borrow fee into consideration when validating the borrow amount.

Go back to Findings Summary

W5: Liquidation instruction causes panic due to unwrap on None value

Impact:	Warning	Likelihood:	N/A
Target:	programs/kamino-lending/src/state/liquidation_operations.rs	Type:	Arithmetics

Description

The issue was discovered running the `usdc-sol` fuzzing session.

Fuzz testing discovered that the liquidate obligation instruction can unexpectedly revert due to calling unwrap on a `None` value. The panicking behavior is triggered when the `user_ltv` is greater than the `max_allowed_ltv`. The following code snippet shows the logic where the panic occurs; specifically, the line `user_ltv.to_percent::<u64>().unwrap()` triggers the panic.

Listing 5. Excerpt from programs/kamino-lending/src/state/liquidation_operations.rs

```
pub fn check_liquidate_obligation(
    &LiquidationCheckInputs {
        lending_market,
        collateral_reserve,
        debt_reserve,
        obligation,
        max_allowed_ltv_override_pct_opt,
        ..
    }: &LiquidationCheckInputs,
) -> Option<LiquidationParams> {
    let user_ltv = obligation.loan_to_value();
    let user_no_bf_ltv = obligation.no_bf_loan_to_value();
    let max_allowed_ltv_user = obligation.unhealthy_loan_to_value();
    let max_allowed_ltv_override_opt =
max_allowed_ltv_override_pct_opt.map(Fraction::from_percent);
    let max_allowed_ltv =
```

```
    max_allowed_ltv_override_opt.unwrap_or(max_allowed_ltv_user);
16
17
18      if user_ltv >= max_allowed_ltv {
19          xmsg!("Obligation is eligible for liquidation, borrowed value
    (scaled): {}, unhealthy borrow value (scaled): {}, LTV: {}%/{}%,
    max_allowed_ltv_user {}%, max_allowed_ltv_override {:?}%",
20
    Fraction::from_bits(obligation.borrow_factor_adjusted_debt_value_sf).to_displ
    ay(),
21
    Fraction::from_bits(obligation.unhealthy_borrow_value_sf).to_display(),
22              user_ltv.to_percent::<u64>().unwrap(),  // panic occurs here
23              max_allowed_ltv.to_percent::<u64>().unwrap(),
24              max_allowed_ltv_user.to_percent::<u64>().unwrap(),
25              max_allowed_ltv_override_pct_opt,
26          );
27
28          // ....
29      }
30  }
31
32
33  impl FractionExtra for Fraction {
34      #[inline]
35      fn to_percent<Dst: FromFixed>(&self) -> Option<Dst> {
36          (self * 100).round().checked_to_num()
37      }
38  }
```

Recommendation

Investigate the panicking behavior and resolve the undesired revert by handling the unwrap operation gracefully.

Fix 2.1

The issue was fixed by additing checked multiplication, preventing the overflow.

Go back to Findings Summary

W6: Withdraw obligation collateral instruction reverts due to immutable owner

Impact:	Warning	Likelihood:	N/A
Target:	programs/kamino-lending/src/handlers/handler_withdraw_obligation_collateral.rs	Type:	Logic error

Description

The issue was discovered running the `usdc-sol` fuzzing session.

Fuzz testing discovered that the Withdraw obligation collateral instruction can unexpectedly revert because the owner is not marked as mutable. The instruction can close the obligation and return the rent to the owner; however, the owner is not explicitly marked as mutable. In cases where the owner is not the transaction fee payer (thus not mutable by default), the transaction will revert because it will not be possible to update the owner's balance.

The following code snippet shows context of the instruction, where the owner is not marked as mutable.

Listing 6. Excerpt programs/kamino-lending/src/handlers/handler_withdraw_obligation_collateral.rs

```
#[derive(Accounts)]
pub struct WithdrawObligationCollateral<'info> {
    pub owner: Signer<'info>,

    // ...
}
```

Later in the instruction, the owner is used to claim the rent from closing the

obligation.

Listing 7. Excerpt programs/kamino-lending/src/handlers/handler_withdraw_obligation_collateral.rs

```
close_account_loader(close_obligation, &accounts.owner, &accounts.obligation)?;
```

Recommendation

Mark the owner as mutable.

Fix 2.1

The issue was fixed by marking the owner as mutable.

Go back to Findings Summary

W7: Instructions cause panic due to division by zero when deposited value is zero

Impact:	Warning	Likelihood:	N/A
Target:	programs/kamino-lending/src/state/obligation.rs	Type:	Arithmetics

Description

The issue was discovered running the `sol-eth-composite` fuzzing session.

Fuzz testing discovered that the repay, withdraw, and redeem instructions can unexpectedly revert due to panicking behavior. The instructions execute the method call `initial_ltv = obligation.loan_to_value();`. The function executed can be seen in the following code snippet. The function panics due to `self.deposited_value_sf` being 0, while the `borrow_factor_adjusted_debt_value_sf` is not 0, causing division by zero.

Listing 8. Excerpt from programs/kamino-lending/src/state/obligation.rs

```
    /// Calculate the current ratio of borrowed value to deposited value
    pub fn loan_to_value(&self) -> Fraction {
        Fraction::from_bits(self.borrow_factor_adjusted_debt_value_sf)
            / Fraction::from_bits(self.deposited_value_sf)
    }
```

Recommendation

Investigate the panicking behavior and resolve the undesired revert by handling the division by zero gracefully.

Fix 2.1

The issue was fixed by not allowing withdraw and repay when the deposited

value is zero.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain Security, Audit Report | Kamino Lend: Fuzz tests, 22.9.2025.

Appendix B: Trident Findings

This section details the results and methodology of fuzz testing performed using the Trident framework during the audit.

B.1. Implementation Details

Fuzz testing with Trident executes the actual Solana program logic without approximations or simplifications. The programs are compiled in their production form and deployed to TridentSVM, ensuring that test results accurately reflect real-world behavior.

Execution Flows represent the core testing strategy:

- **Revision 1.0**: Used predefined sequences of Solana instructions executed during the fuzz testing process, following specific user scenarios and administrative workflows
- **Revision 2.0**: Implemented random Solana instruction selection with variations and repetition (ordered sampling with replacement), allowing discovery of unexpected account state combinations

Testing Environment Considerations:

- **Price Oracle Integration**: Scope prices were dumped from Solana Mainnet before each fuzz testing session to ensure tests used current, realistic asset prices
- **Liquidation Testing**: To test liquidation scenarios, the index of asset into scope price within reserve is modified to point to the asset that has lower price
- **Multi-User Scenarios**: Each fuzz test has two reserves. Alice deposits to reserveA, obtains collateralA, Bob deposits to reserveB, then Alice uses collateralA to borrow reserveB

B.2. Fuzz testing

The following table lists all implemented execution flows in the Trident fuzz testing framework.

ID	Flow	Added
F1	Test the admin-related instructions	1.0
F2	Test the user related methods utilizing the Main Lending Market config and SOL reserve config	1.0
F3	Test the user related methods utilizing the Main Lending Market config and cbBTC reserve config	1.0
F4	Test the user related methods utilizing the Main Lending Market config and ETH reserve config	1.0
F5	Test the user related composite methods utilizing the Main Lending Market config and SOL reserve config	1.0
F6	Test random instruction selection	1.0

Table 4. Trident fuzz testing flows - revision 1.0

ID	Flow	Added
F7	Initialization instructions, Lending Market initialization, Obligation initialization etc. (Ensure states are properly initialized and there is enough liquidity in the reserves)	2.0
F8	Deposit reserve liquidity instruction and refresh reserve liquidity, borrow liquidity and obligation	2.0
F9	Withdraw collateral from obligation and refresh reserve liquidity, borrow liquidity and obligation	2.0
F10	Deposit collateral into obligation and refresh reserve liquidity, borrow liquidity and obligation	2.0

ID	Flow	Added
F11	Borrow obligation liquidity, forward 100k to 1m slots and refresh reserve liquidity, borrow liquidity and obligation	2.0
F12	Repay obligation liquidity and refresh reserve liquidity, borrow liquidity and obligation	2.0
F13	Liquidate obligation and redeem reserve collateral and refresh reserve liquidity, borrow liquidity and obligation	2.0
F14	Redeem reserve collateral and refresh reserve liquidity, borrow liquidity and obligation	2.0
F15	Mark obligation for deleveraging and refresh reserve liquidity, borrow liquidity and obligation	2.0
F16	Terminating instructions, Redeem Fees and Withdrawal Fees	2.0

Table 5. Trident fuzz testing flows for simple instructions - revision 2.0

ID	Flow	Added
F17	Initialization instructions, Lending Market initialization, Obligation initialization etc. (Ensure states are properly initialized and there is enough liquidity in the reserves)	2.0
F18	Deposit reserve liquidity and Obligation collateral instruction, and refresh reserve liquidity, borrow liquidity and obligation	2.0
F19	Withdraw collateral from obligation and redeem reserve collateral, and refresh reserve liquidity, borrow liquidity and obligation	2.0

ID	Flow	Added
F20	Repay and Withdraw and Redeem, and refresh reserve liquidity, borrow liquidity and obligation	2.0
F21	Repay and Withdraw and Redeem, and refresh reserve liquidity, borrow liquidity and obligation	2.0
F22	Repay and Withdraw and Redeem, and refresh reserve liquidity, borrow liquidity and obligation	2.0
F23	Deposit and Withdraw, and refresh reserve liquidity, borrow liquidity and obligation	2.0
F24	Flash borrow and Flash repay, and refresh reserve liquidity, borrow liquidity and obligation	2.0
F25	Liquidate obligation and redeem reserve collateral and refresh reserve liquidity, borrow liquidity and obligation	2.0
F26	Terminating instructions, Redeem Fees and Withdrawal Fees	2.0

Table 6. Trident fuzz testing flows for composite instructions - revision 2.0

ID	Flow	Added
F27	Initialization of Lending Market	2.0
F28	Update Lending Market (randomly selecting the option to update)	2.0
F29	Update Lending Market Owner	2.0

Table 7. Trident fuzz testing flows for lending market update instructions - revision 2.0

The following tables list all implemented invariant checks in the Trident fuzz testing framework.

ID	Invariant	Added	Status
IV1	Instruction reverts with unhandled panic	1.0	**Fail** (W1, W2)
IV2	The Lending Market is initialized as expected	1.0	Success
IV3	The Reserve is initialized as expected	1.0	Success
IV4	The Farms for Reserve are initialized as expected	1.0	Success
IV5	Updating Lending Market can be performed only by the authority and updates are performed as expected	1.0	Success
IV6	Updating Reserve config can be performed only by the authority and updates are performed as expected	1.0	Success
IV7	New Lending Market owner can be claimed only by the cached owner	1.0	Success
IV8	Redeem and Withdrawal protocol fees works as expected	1.0	Success
IV9	User metadata are initialized as expected	1.0	Success
IV10	Referrer Token State is initialized as expected	1.0	Success
IV11	Deposit reserve liquidity works as expected	1.0	Success
IV12	Redeem reserve collateral works as expected	1.0	Success
IV13	Obligation is initialized as expected	1.0	Success
IV14	Deposit obligation collateral works as expected	1.0	Success

Table 8. Trident fuzz testing invariants - revision 1.0

ID	Invariant	Added	Status
IV15	Instruction reverts with unhandled panic or unexpected revert	2.0	**Fail** (W5, W6, W7)
IV16	Borrow liquidity: User receives exactly the same amount as requested	2.0	Success
IV17	Borrow liquidity: User requested borrow amount + fees is transferred from reserve	2.0	Success
IV18	Borrow liquidity: Borrow amount + fees is not greater than the borrow limit	2.0	**Fail** (W4)
IV19	Deposit Collateral into obligation: Collateral amount is transferred to the vault	2.0	Success
IV20	Deposit Collateral into obligation: Obligation collateral is updated with correct liquidity amount	2.0	Success
IV21	Deposit Collateral into obligation: Amount transferred from user to reserve is as expected	2.0	Success
IV22	Deposit Collateral into obligation: Correct amount of collateral is immediately transferred to the vault	2.0	Success
IV23	Deposit Liquidity and Collateral: Supply of collateral correctly increases	2.0	Success
IV24	Deposit Liquidity and Collateral: Obligation collateral is updated with correct liquidity amount	2.0	Success
IV25	Deposit Liquidity: Correct amount is transferred from user to reserve	2.0	Success

ID	Invariant	Added	Status
IV26	Deposit Liquidity: Correct amount of collateral is minted to the user	2.0	Success
IV27	Deposit Liquidity: The collateral supply increases just by the amount minted to the user	2.0	Success
IV28	Flash Borrow and Repay: Expected protocol fee is transferred from user	2.0	Success
IV29	Flash Borrow and Repay: Reserve contains the same amount as before the flash borrow	2.0	Success
IV30	Flash Borrow and Repay: Expected fee is added to the fee receiver	2.0	Success
IV31	Redeem Reserve Collateral: User transfers out the expected collateral amount	2.0	Success
IV32	Redeem Reserve Collateral: Expected redeem amount is transferred from reserve to user	2.0	Success
IV33	Redeem Reserve Collateral: The amount transferred from the reserve is exactly the same as the amount transferred to the user	2.0	Success
IV34	Redeem Reserve Collateral: Supply of the collateral is decreased by the amount redeemed	2.0	Success
IV35	Redeem Fees: Expected fees are transferred to the fee receiver	2.0	Success
IV36	Redeem Fees: Expected fees are subtracted from the reserve available liquidity	2.0	Success
IV37	Repay Obligation: Reserve liquidity is increased by the amount repaid	2.0	Success

ID	Invariant	Added	Status
IV38	Repay Obligation: Obligation borrow amount is correctly decreased by the expected settle amount	2.0	Success
IV39	Withdraw Collateral: User receives expected amount of collateral as requested	2.0	Success
IV40	Withdraw Collateral: The amount of collateral transferred from the reserve is the same as the user requested	2.0	Success
IV41	Withdraw Collateral: The Obligation deposited amount is correctly subtracted by the corresponding liquidity amount	2.0	Success
IV42	Withdraw Protocol fees: Expected amount is transferred from the fee vault	2.0	Success
IV43	Withdraw Protocol fees: The amount transferred from the fee vault is the same as the amount received by the fee receiver	2.0	Success

Table 9. Trident fuzz testing invariants - revision 2.0

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz