Security Assessment & Formal Verification Final Report

Kamino Vault

April 2025

Prepared for Kamino Finance

certora

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
KVault	https://github.com/Kamino-Finance/kvault-private	`ee552e9`	Solana

Project Overview

This document describes the specification and verification of **Kamino Vault Protocol** using the **Certora Prover**, as well as **manual code review** findings. The work was undertaken from `Jan 7, 2025` to `Jan 23, 2025` as well as `Apr 2, 2025` to `Apr 25, 2025`

All contracts in the following folder are included in our scope:

`kvault-private/programs/kamino-vault/src/*`

The Certora Prover demonstrated that the implementation of the Solana contracts above is correct (modulo identified issues) with respect to the formal properties formulated and written by the Certora team. In addition, the team performed a manual audit of all the Solana contracts. During the verification process and the manual audit, the Certora team discovered bugs in the protocol code, as listed on the following pages.

Protocol Overview

The Kamino Lending platform (KLend) is a decentralized finance (DeFi) platform designed to optimize the borrowing and lending of crypto assets on the Solana blockchain. Amongst other things, the platform enables users to lend assets, that is to deposit cryptocurrencies into liquidity pools to earn interest.

KLend supports multiple markets, each with different interest rates and parameters.
To achieve the optimal risk/reward ratio a user would need to divide his assets and manually invest in different markets. Based on market conditions and borrow rates, the risks and rewards for a market could change and the user would need to move his assets between markets to maintain the same level of security or earnings.

Kamino Vault (KVault) offers an easy way to lend assets in multiple KLend markets. A vault can be configured to allocate assets over multiple KLend markets. The ratio of how much is invested in which vault is managed by the vault admin. A vault admin has the possibility to change the allocation weights and specify how much of the vault's assets are invested in which reserves to maintain a specific risk/reward profile.

Users can deposit into the vault and have their assets invested into the KLend reserves, based on the configured allocations. Assets managed by the vault can be rebalanced between the different reserves to maintain the configured allocation distribution.
The vault can charge a management fee (based on amount deposited and time) and a performance fee (percentage of KLend earnings).

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	0	-	-
High	0	-	-
Medium	3	3	3
Low	2	2	2
Informational	7	-	4
Total	10		

Severity Matrix

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

certora

Detailed Findings

ID	Title	Severity	Status
M-01	give_up_pending_fee() will not work when pending_fees > AUM	Medium	Fixed in 99bea9f
M-02	A first depositor could manipulate the share price in order to favor from rounding errors	Medium	Fixed in 1c3b089
M-03	Rounding error due to collateral_exchange_rate	Medium	Fixed in 92c9c5c (kamino-lending)
L-01	Incorrect enforcement of allocation cap	Low	Fixed in b11b6f0
L-02	Vault owner can block withdrawals via unbounded min_withdraw_amount	Low	Fixed in 453cc8e
L-03	Imprecise cToken calculation during withdrawal	Low	Fixed in 22828f3
I-01	Unused _reserves_iter variable in handler_withdraw	Informational	Fixed in 6eccacc
I-02	Unneeded pending_fees > Fraction::ZERO check in compute_aum()	Informational	Fixed in e90906e

I-03	min_invest_delay_slots and last_invest_slot not implemented	Informational	Fixed in e90906e
I-04	Confusing variable naming/usage in vault_checks	Informational	Fixed in e90906e
I-05	A call to give_up_pending_fee() could be anticipated and exploited by depositing beforehand	Informational	Will not be fixed
I-06	Losses could or could not offset gains for the calculation of the performance fees based on if charge_fees() was called or not	Informational	Will not be fixed
I-07	Discrepancy of share rate calculation	Informational	Will not be fixed

Medium Severity Issues

M-01 give_up_pending_fee() throws error when pending_fees > AUM

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: vault_operations.rs	Status: Fixed in Commit 99bea9f	Violated rule: `give_up_pending_fees_no_revert`

Description: When the vault has invested in reserves that experience socialized losses, there is a rare scenario that allows the pending fees in the vault to be bigger than the actual assets under management.
This will put the vault in a state where it is effectively locked and most instructions return an `AUMBelowPendingFees` Error.
The `give_up_pending_fee()` instruction was added to allow the vault owner to give up his fees and return them to the vault to get the vault back in a normal working state.
From the `give_up_pending_fee` function, there is a call to `charge_fees`, which uses the `compute_aum` function. The check for `AUM < pending_fees` is done in `compute_aum`, resulting in an `AUMBelowPendingFees` error.
Thus, `give_up_pending_fee` cannot be used to get the vault in a working state again and any remaining value is locked in the vault.

Recommendations:
We recommend removing the `charge_fees` calls from `give_up_pending_fee`. This could potentially result in missing some management fees, but since the intention is to give up these fees, this should not be an issue.

Customer's response: Confirmed, fixed in commit 99bea9f

Fix Review: The `charge_fees` function now handles the case where `give_up_pending_fee` returns an error. In this case it sets the aum to zero. This correctly fixes the issue.

M-02 A first depositor could manipulate the share price in order to favor from rounding errors		
Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: vault_operations.rs	Status: Fixed in commit 1c3b089	

Description: A first depositor can inflate the share price by repeatedly depositing and withdrawing funds. Once the share price is high enough, any future deposits of amounts smaller than the share price will fail (`shares_to_mint` cannot be 0), and any deposits slightly larger than that might cause the depositor to lose a significant amount due to rounding.

A proposed mechanism to exponentially inflate the share price from an initial state of 1 share and 2 AUM – this state is reachable by depositing some initial amount, waiting for the AUM to rise due to a profit, and then withdrawing all shares but one:

A test case exemplifying this issue can be found in Appendix A.

Recommendations:
Implement one of the standard ways to protect against First depositor attacks. See OpenZeppelin solution for example.

Customer's response: Fixed in 1c3b089

Fix Review: On vault initialization it is now required to supply an initial deposit of 1000 lamports. Shares for this deposit are accounted for, but no shares are actually minted, making it impossible to remove the initial deposit.This initial deposit significantly increases the cost for this attack while also lowering the profits. These effects are enough to effectively prevent this type of attack.

M-03 Rounding error due to collateral_exchange_rate: total supply of reserve can be exceeded when withdrawing from reserve

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: vault_operations.rs	Status: Fixed in commit 92c9c5c (kamino-lending)	Violated rules: rounding_error_ctoken

Description: When withdrawing funds from the vault, it might trigger a withdrawal from some invested reserve to cover additional funds to send to the user not currently available in the vault. For this computation, first the amount of cTokens required to cover the liquidity that needs to be taken from the reserve is allocated via `exchange_rate.fraction_liquidity_to_collateral()`.
Then the resulting amount of liquidity from withdrawing these cTokens is calculated by `exchange_rate.fraction_collateral_to_liquidity(invested_to_disinvest_ctokens_f)`.

Relevant Code:

```
Unset
let invested_to_disinvest_ctokens_f = exchange_rate
       .fraction_liquidity_to_collateral(invested_liquidity_to_send_to_user_f)
       .ceil();


let invested_to_disinvest_ctokens: u64 = invested_to_disinvest_ctokens_f.to_num();

// compute the expected total liquidity disinvested as we can withdraw slightly less than the
user is entitled to so we make sure we have enough for the withdrawal

let invested_liquidity_to_disinvest_f = exchange_rate
       .fraction_collateral_to_liquidity(invested_to_disinvest_ctokens_f);

let invested_liquidity_to_disinvest = invested_liquidity_to_disinvest_f.to_floor::<u64>();
```

As is indicated in the counterexample provided by the Certora prover in rule `rounding_error_ctoken_respects_liquidity_supply` the double division by the `collateral_exchange_rate` which is a fixed point number in this computation can lead to rounding errors such that the calculated liquidity in `invested_liquidity_to_disinvest_f` can exceed the reserve's total supply.

Recommendations: Implement a standard way of using mulDiv to calculate the rate between shares and liquidity, rather than reusing the `collateral_exchange_rate`.

Customer's response: Fixed in kamino-lending commit 92c9c5c

Fix Review: The `CollateralExchangeRate` implementation in kamino-lending has been updated to use mulDiv for all calculations. Because of this change, the vault can now safely use `collateral_exchange_rate` for the calculation and no further changes are needed to resolve this issue.

Low Severity Issues

L-01 Incorrect enforcement of allocation cap		
Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: state.rs	Status: Fixed in b11b6f0	

Description: A vault admin can set weight and cap for allocations.
There is an issue with the way the allocation targets are calculated that allows for allocations to get a target amount larger than the cap.

`refresh_target_allocations()` sets the target amounts for all allocations before investing. It does so by iterating over all allocations and setting target amounts based on allocation weight/total weight.

If an amount for an allocation is above the cap, the target is set to the capped amount and the remainder is split between other allocations in the next iterations.

The check for the cap is done over the amount to add instead of the total amount for the allocation. This can cause issues when there are multiple allocations with a cap.

Exploit Scenario:

```
allocation1: weigth 45 cap 100k
allocation2: weigth 55 cap 100k
aum = 222k

First iteration:

allocation1: 99.9k  (222k * 45%)
allocation2: 100k   (222k * 55% = 122.1k capped to 100k,22.1k remaining_tokens_to_allocate)
```

```
Second iteration, only allocation1

22.1k < cap, will be added to allocation1

Target amounts will become:

allocation1: 122k
allocation2: 100k
```

According to settings 45% should be invested in allocation1 and 55% in allocation2 with a max of 100k, but there is now almost 55% in allocation1 and the value is above the cap.

Recommendations:
Include the `token_target_allocation` from previous iterations in the capped amount check

Customer's response: Fixed in b11b6f0

Fix Review: Fix correctly resolved the issue

L-02 Vault owner can block withdrawals via unbounded min_withdraw_amount

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: handler_update_vault_config.rs	Status: Fixed in 453cc8e	

Description: The vault configuration has a `min_withdraw_amount` variable. Users are not able to make withdrawals with amounts smaller than `min_withdraw_amount`.
According to Kamino Finance the main reason for this variable is to be able to prevent withdrawals of tiny amounts.

A vault admin is able to change `min_withdraw_amount` via `update_vault_config`. In this process there is no limit to the amount entered. This allows a vault owner to set `min_withdraw_amount` to a large amount to disallow all users to withdraw any deposited funds.

Recommendations: We recommend to specify a maximum value for `min_withdraw_amount` and enforce that in the check of `update_vault_config`.

Customer's response: Fixed in 453cc8e

Fix Review: The fix adds an `UPPER_LIMIT_MIN_WITHDRAW_AMOUNT` of 1000 which correctly prevents admin misuse of the `min_withdraw_amount` to block withdrawals.

L-03 Imprecise cToken calculation during withdrawal

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: vault_operations.rs	Status: Fixed	Violated rules: rule_ctoken_rounding_match_user_needs

Description: When a user withdraws funds from the vault, the system may need to retrieve liquidity from an invested reserve to fulfill the user's full entitlement. To determine how many cTokens must be disinvested to cover the required liquidity, the following computation is used:

```
Unset
let invested_to_disinvest_ctokens_f = exchange_rate
       .fraction_liquidity_to_collateral(invested_liquidity_to_send_to_user_f)
       .ceil();
```

However, the call to `.ceil()` has no effect in this context. This is because `fraction_liquidity_to_collateral` performs a flooring division internally, making the subsequent ceiling operation redundant. As a result, the computed number of cTokens may be insufficient to fully cover the intended liquidity amount, especially when `invested_liquidity_to_send_to_user_f` includes a fractional value. In such cases, a small shortfall may occur due to the unaccounted fractional portion.

Recommendations:
We recommend implementing a `fraction_liquidity_to_collateral_ceil` complementary to the already existing `collateral_to_liquidity_ceil`.

Customer's response: Fixed in 22828f3

Fix Review: The implementation of `fraction_liquidity_to_collateral_ceil` in combination with tracking the allocated cTokens per reserve allocation safely fixes this issue.

Informational Severity Issues

I-01. Unused _reserves_iter variable in handler_withdraw

Description: withdraw_utils::withdraw in handler_withdraw.rs has the following code block

```
Unset
let (reserve_address_to_withdraw_from, reserve_state_to_withdraw_from, _reserves_iter): (
	_,
	_,
	Box<dyn Iterator<Item = FatAccountLoader<Reserve>>>,
) = if should_withdraw_from_invested {
	let withdraw_from_reserve_accounts = &ctx_withdraw_from_reserves.unwrap();
	let reserve = withdraw_from_reserve_accounts.reserve.load()?;
	let reserve_address = withdraw_from_reserve_accounts.reserve.to_account_info().key;
	let reserve_allocation = vault_state
		.allocation_for_reserve(&withdraw_from_reserve_accounts.reserve.key())?;
	require_keys_eq!(
		reserve_allocation.ctoken_vault,
		withdraw_from_reserve_accounts.ctoken_vault.key()
	);

	let reserves_iter = remaining_accounts
		.iter()
		.take(reserves_count)
		.map(|account_info|
		FatAccountLoader::<Reserve>::try_from(account_info).unwrap());
	(
		Some(reserve_address),
		Some(reserve),
		Box::new(reserves_iter),
	)
} else {
	(None, None, Box::new(iter::empty::<_>()))
};
```

The `_reserves_iter` variable is never used, which makes a large part of this code redundant.

Recommendation: Remove the inner `reserves_iter` variable and have the block only return `reserve_address` and `reserve`.

Customer's response: Fixed in 6eccacc

I-02 pending_fees > Fraction::ZERO check in compute_aum() redundant

Description: In the `compute_aum` function in state.rs we have the following block of code:

```
Unset
if Fraction::from(self.token_available) + invested.total < pending_fees
       && pending_fees > Fraction::ZERO
{
       return err!(KaminoVaultError::AUMBelowPendingFees);
}
```

This checks if the sum of `token_available` and `invested.total` is less than pending fees and pending fees bigger than 0.
`token_available` and `invested.total` are both unsigned.
If the sum of two unsigned numbers is less than `pending_fees`, then the pending fees always need to be bigger than 0.
This means the second part (`&& pending_fees > Fraction::ZERO`) will always be true when the first part is true.

Recommendation: Consider removing `&& pending_fees > Fraction::ZERO` part

Customer's response: Fixed in e90906e

I-03 min_invest_delay_slots and last_invest_slot not implemented

Description: vault state has a `min_invest_delay_slots` variable which seems to be intended to set a minimum delay between two invest instructions. The vault allocation has a `last_invest_slot` to keep track of when the last invest was done for this allocation.
Both variables are not used. The last invest slot is not updated and minimum delay slots are not enforced.

A vault owner is able to set a value for `min_invest_delay_slots` via `update_vault_config`.
Being able to set a `min_invest_delay_slots` setting, can give the vault owner the impression that this restriction will be enforced.

Recommendation: Implement the investment delay check or remove the variables (replace by padding) to make clear that this functionality is not available.
Alternatively, disable updating this variable via `update_vault_config`.

Customer's response: Fixed in e90906e

I-04 Confusing variable naming/usage in vault_checks

Description: In vault_checks.rs the functions `post_transfer_withdraw_balance_checks` and `post_transfer_withdraw_pending_fees_balance_checks` check if all fund transfers are correct.

They use local variables to represent the amount differences for various accounts. For most of these variables, it uses "**<account_name>_diff**" as the variable name and `i128` type.

Even though the variables use a signed integer type, they do not follow the usual “positive diff is increment” and “negative diff is decrement” pattern. This makes it harder and less intuitive to read and understand the checks.

Recommendation: To improve code readability, consider following the usual pattern where an increment is represented by a positive diff value and decrement by a negative diff value and updating the corresponding require_msg .

Alternatively the variable names could be renamed to indicate the direction, e.g.

```
Unset

let token_vault_diff: i128 = i128::from(amounts_before.vault_token_balance)
      - i128::from(amounts_after.vault_token_balance);
```

where a positive `token_vault_diff` indicates a decrease of `token_vault` balance would enhance readability by

```
Unset
let token_vault_decrease: i128 = i128::from(amounts_before.vault_token_balance)
      - i128::from(amounts_after.vault_token_balance);
```

Customer’s response: In most cases, we believe it's more intuitive for variable names to include `token_vault_diff`, as it suggests they represent the change that occurred in the operation. We've updated several variable names to reflect the only possible sign/direction of the operation in commit e90906e.

I-05 A call to give_up_pending_fee() could be anticipated and exploited by depositing beforehand

Description: Anyone who can anticipate that the vault's owner is about to call `give_up_pending_fee()` can take advantage of that fact by depositing a large amount into the vault in order to hold a large amount of the shares. As the amount of fees that is being forfeited is divided proportionally between the shareholders, this mechanism could be misused in order to gain more on the expense of other shareholders.

Recommendation: Consider whether that is acceptable. Fixing this would likely require some major code changes.

Customer's response: Acknowledged. The purpose of `give_up_pending_fee()` is mostly for emergency cases and this is not a critical issue that we plan to fix.

I-06 Losses could or could not offset gains for the calculation of the performance fees based on if charge_fees() was called or not.

Description: As performance fees are calculated globally and not per user at the time of withdrawal, the amount of performance fees that will be taken in the case where there are both periods of losses and periods of gains could depend on whether or not the charge_fees() function was called in between. Say there was a period where the AUM dropped followed by a period that the AUM rose back to the original amount. If charge_fees() would be called after the period of loss, then that would update the state and make it such that the next period of gains would be taxed. In contrast, if charge_fees() was not called, then the losses will just offset the gains and no performance fees will be taken.

Recommendation: Consider whether that is acceptable. Fixing this would likely require some major code changes.

Customer's response: Acknowledged. The AUM going down is a very unlikely event (as it would require a socialized loss event on a reserve where the vault invested), this is an accepted behavior from our side.

I-07 Discrepancy of share rate calculation

Description: There is discrepancy in the share-to-liquidity and liquidity-to-share conversion rates during withdrawal process as shown by the counterexample in rule `rule_shares_to_burn_integrity`.
The counterexample arises when comparing two different withdrawals. In two withdrawal scenarios with the same amount of liquidity being sent to the user (due to availability), a counterintuitive behavior occurs: the withdrawal with a higher number of shares will burn more shares, despite transferring an identical amount of liquidity. This means users with more shares could incur a disproportionate share-burning penalty, even when the actual liquidity transfer remains constant.

The share conversion rates differ between two critical functions:

1. `compute_user_total_received_on_withdraw()` uses the actual rate: `shares_issued / current_aum`
2. `calculate_shares_to_burn()` uses: `shares_issued / current_aum.floor()`

This can create an inconsistency for the user in shares that are burned during the withdrawal process when the available amounts sent to the user are equal but the original number of shares to withdraw differs.
While the calculation is still in favor of the protocol, it might lead to unexpected behavior from the user perspective in cases where not enough funds are available to disinvest from a reserve to cover the whole withdrawal.

Recommendation: Standardize the conversion rate to `shares_issued / current_aum.floor()` which still favors the protocol but ensures consistent share conversion across functions.

Customer's response: This can lead to 1 lamport of share loss from the withdrawer but help in scenarios like withdrawing their whole position and we think this is a desirable behavior. No action.

Formal Verification

Methodology

Rules: A rule is a verification task possibly containing assumptions, calls to the relevant functionality that is symbolically executed and assertions that are verified on any resulting states from the computation.

Inductive Invariants: Inductive invariants are proved by induction on the structure of the smart contract under verification. We use constructors/init() functionality as a base case, and consider all other (relevant) externally callable functions as step cases.
Specifically, to prove the base case, we show that a property holds in any resulting state after a symbolic call to the respective initialization function. For proving step cases, we generally assume a state where the invariant holds (induction hypothesis), symbolically execute the functionality under investigation, and prove that after its computation any resulting state satisfies the invariant. Each such case is performed in one rule.

Note that to make verification more tractable, we sometimes prove on lower level functions that contain the relevant logic. In the case of Kamino, we prove invariants correct by proving properties on the relevant functionality provided in `vault_operations.rs`.

General Assumptions and Simplifications

Configuration and Munging

1) For tractability, we chose an **approximation of the `Fraction` datatype**. This datatype is used to represent fixed point numbers in Solana and is a wrapper for Rust library `fixed` using `FixedU128<60>`. The code is based on `Fraction128<60>` where 68 bits represent the integer part of the number and 60 are the fractional bits. We use fractional numbers of type `FixedU64<14>` with 50 bits used for the integer and 14 bits used for the fractional part. This makes the verification task simpler on the verification engine and allows us to catch potential rounding errors.

2) Loops are inherently difficult for formal verification. We handle loops by unrolling them a specific amount of times. We thus use an **underapproximation on the max_reserves of**

1. In our setting, an obligation position contains at most two different deposits and one borrow. Consequently, we use a **loop_iter of 1**, unrolling each loop exactly once.

3) Specifically, to make sure that nondetermistic values of reserve holdings match throughout computations, we summarize the holdings computation of the vault in `amounts_invested()` to base the liquidity in reserves on the contents of the first `vault_allocation_strategy`. Along these lines, we summarize
    a) `in_reserve()` to return the first `vault_allocation_strategy` and assume matching reserve pubkeys,
    b) `deposit_into_vault_allocation()` to increase the first allocation's cToken amount and assume matching reserve pubkeys,
    c) `withdraw_from_vault_allocation()` to decrease the first allocation's cToken amount and assume matching reserve pubkeys,
    d) `get_reserves_with_allocation_count()` to return at most one if an allocation is set and
    e) `set_allocation_last_invest_slot()` to adapt the last invest slot on the first allocation strategy and assume matching reserve pubkeys.
4) We underapproximate the `collateral_exchange_rate` with `Fraction::ONE` to make verification tractable. This change affects computations in `amounts_invested()`, `withdraw()`, `withdraw_pending_fees()` and `invest()`.
5) To prove correctness of functionality independently of fee charges, we adapt `charge_fees()` to not compute new fees based on the current timestamp being the `last_fee_charge_timestamp` stored in the vault state.

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue.
Violated	A counterexample exists that violates one of the assertions of the property.

Formal Verification Properties

Vault Solvency

Module Properties

To check correctness of the Kamino Vault's accounting, we prove **three main invariants**:

P1 **AUM cannot decrease**: apart from charging fees and withdrawing liquidity, the assets under management cannot decrease.
P2 **Solvency**: number of minted shares ≤ AUM. In other words, the amount of shares minted is accounted for by assets under management.
P3 **Share value does not decrease:** no vault operation can decrease the share value.

Module Assumptions

To check correctness of the above properties on vault operations:

- we assume fees are already charged and prove properties on `charge_fees()` separately
- We assume a `collateral_exchange_rate` of `Fraction::ONE` to make verification tractable.

P-01. Vault Solvency	
Status: Verified	Specification: We prove P1-P3 for all public vault operations. The following rules assume fees to be already charged.

Rule Name	Status	Description	Link to rule report
rule_solvency_initialize	Verified	*A newly initialized vault respects P1-P3:* *Note we prove that vault.shares_issued and aum are 0, the properties trivially follow.*	*Summarized Report*

rule_solvency_deposit	Verified	*Depositing into the vault respects P1 (no aum decrease) and P2 (solvency).* *Additionally, we check that invested holdings do not change on deposits, all minted shares are accounted for in the total shares supply, prev_aum is updated according to the deposit and aum can change by at most the max_deposit_amount.*
rule_share_value_deposit	Verified	*Depositing into the vault does not decrease share value (P3).* *We assume P1 and P2, fees to be charged and use lemma_shares_to_mint proved in rule_lemma_shares_to_mint*
rule_lemma_shares_to_mint	Verified	*We prove that get_shares_to_mint respects P1-P3.*
rule_aum_decrease_withdraw	Verified	*We prove P1 for withdraw() by showing that AUM can only decrease by the total calculated withdraw amount. Additionally we check that all burnt shares are correctly deducted from the total shares supply, as well*
rule_solvency_withdraw_case1	Verified	*We prove P2-P3 for withdraw()* *Case1: shares_to_burn < shares_to_withdraw,* *- requires lemma_case1* *- proved in rule_lemma_proof_case1*
rule_lemma_proof_case1	Verified	*We prove that calculate_shares_to_burn respects P2 for shares_to_burn < shares_to_withdraw*

		Note showing P2 is sufficient to show P2 and P3 in the main rule.	
rule_solvency_withdraw_case2	Verified	*We prove P2-P3 for withdraw()* *Case1: shares_to_burn == shares_to_withdraw,* *- requires lemma_case1* *- proved in rule_lemma_proof_case1*	
rule_lemma_proof_case2	Verified	*We prove that calculate_shares_to_burn respects P2 and P3 for shares_to_burn == shares_to_withdraw*	
rule_solvency_withdraw_pending_fees	Verified	*We prove P1-P3 for withdraw_pending_fees. Additionally, we prove that shares_issued is not impacted by this computation.*	
rule_solvency_invest	Verified	*We prove P1-P3 for invest. Additionally, we prove that shares_issued is not impacted by this computation.*	
rule_solvency_give_up_pending_fees	Verified	*We prove P1-P3 for give_up_pending_fees. Additionally, we prove that shares_issued is not impacted by this computation, aum is increased at least by the amount of given up fees and fees are either zeroed out or correctly reduced by the amount that was given up.*	

rule_solvency_update_reserve_allocation	Verified	*We prove P1-P3 for invest. Additionally, we prove that shares_issued is not impacted by this computation.*	

P-02. Solvency for charge_fees

Status: Verified	Specification: We prove that charge_fees() respects P1 AUM does not decrease, as well as P2 Solvency. Note that in order to prove correctness of these properties, we adapted the code to require that `mgmt_fee <= (earned_interest - prf_fee),` that is the charged management fee cannot exceed any earned interest between computations. Otherwise, we cannot prove these properties correct as `mgmt_fee` can get arbitrarily large. We additionally prove a weaker form of P3 that only checks that the ratio of total holdings, that is AUM without fees taken into account, to shares does not decrease upon fee charging

Rule Name	Status	Description	Link to rule report
rule_solvency_charge_fees	Verified	*We prove P1 and P2 for charge_fees().* *Note that this requires the assumption that mgmt_fee <= (earned_interest - prf_fee).*	Summarized Report
rule_share_value_charge_fees	Verified	*We prove an alternate version of P3: holdings.total_sum/shares does not decrease by charge_fees().*	

P-03. give_up_pending_fees does not revert

Status: Verified after fix	Specification: We prove that give_up_pending_fees decreases fees accordingly and does not revert in its main use case.

Rule Name	Status	Description	Link to rule report
rule_give_up_pending_fees_integrity	Verified	*We prove that give_up_pending_fees correctly reduces the fees and does not decrease prev_aum.*	*Summarized Report*
rule_give_up_pending_fees_no_revert	Verified after fix	*We prove that give_up_pending_fees does not revert when fees exceed holdings which is the main use case of this function. See issue M-01*	

P-04. Integrity of disinvesting ctokens for withdrawals

Status: Verified after fix	Specification: We prove that withdrawing invested liquidity from a reserve does not result in unaccounted rounding errors.

Rule Name	Status	Description	Link to rule report
rule_ctoken_rounding_respects_liquidity_supply	Verified after fix	*We prove that when withdrawing invested liquidity from a reserve for user withdrawals, the vault cannot withdraw more liquidity than the reserve's supply.* *A violation is found where due to rounding with collateral_exchange_rate the withdrawn liquidity from the reserve is overvalued such that it exceeds the reserve's total supply. See Issue M-03*	*Summarized Report*
rule_ctoken_rounding_respect_mint_supply	Verified	*We prove that the calculated cToken amount to disinvest respects the reserve's mint supply.*	
rule_ctoken_rounding_match_needs_of_user	Verified after fix	*We prove that the liquidity withdrawn from an invested reserve to cover a user's withdrawal is enough to match the needed funds for the withdrawal. A violation is found where due to division in fraction_collateral_to_liquidity() the withdrawn liquidity from the reserve is insufficient to cover the user's withdrawal. See Issue L-03*	

P-05. shares_to_burn consistency

Status: Violated, **won't fix.** See Issue I-07	Specification: We prove that two calls to withdraw resulting in the same amount of liquidity should burn an equal number of shares.

Rule Name	Status	Description	Link to rule report
rule_shares_to_burn_consistency	Violated	*We prove that when equal liquidity is provided via a withdrawal, then the amount of shares to burn is equal for both calls.* *This is violated by calls that originally contain a higher number of shares to withdraw when the available amount is less than the user wants to withdraw, see Issue I-07*	*Report*

certora

Appendix A

A test case showing M-02 first depositor attack:

Unset

```
--- a/programs/kamino-vault/tests/tests_deposit_invest_withdraw_with_yield.rs
+++ b/programs/kamino-vault/tests/tests_deposit_invest_withdraw_with_yield.rs
@@ -1,5 +1,6 @@
 use ::kamino_vault::program_test::kamino_vault::VaultDepositorAccounts;
-use kamino_lending::{assert_fuzzy_eq, program_test::common::math::usdc_decimal_to_u64};
+use kamino_lending::{assert_fuzzy_eq, assert_gt, program_test::common::math::usdc_decimal_to_u64};
+use kamino_vault::xmsg;
 use kamino_vault::program_test::kamino_vault::{client, fixtures};
 use solana_program_test::tokio;
 use solana_sdk::{pubkey::Pubkey, signer::Signer};


#[tokio::test]
async fn test_vault_first_depositor_bug() {
    let (mut ctx, vault, owner, investor, mkts) =
        fixtures::setup_markets_and_reserves_with_yield(1).await;

    client::init_vault(&mut ctx, &owner, &vault).await.unwrap();
    client::update_reserve_allocation(&mut ctx, &owner, &vault, &mkts[0].reserve(), 100, u64::MAX)
        .await;

    let initial_balance = usdc(100_000_000.0);
    let depositor = VaultDepositorAccounts::new(&mut ctx, &vault, initial_balance).await;
    let reserve = &mkts[0].reserve();

    let deposit_amount = usdc(100.0);
    client::deposit(&mut ctx, &vault, &depositor, deposit_amount).await;
    client::invest(&mut ctx, &investor, &vault, &mkts[0].reserve_accounts()).await;

    xmsg!("Forward time to increase aum from reserve profits");
    ctx.fast_forward_minutes(1).await;

    let mut balances = vault.balances(&mut ctx).await;
    let amount = balances.shares_supply - 1;
    xmsg!("Start withdrawal of {} shares", amount);
    client::withdraw(&mut ctx, &vault, &depositor, amount, reserve).await;

    xmsg!("Balances after withdraw");
    let mut aum = vault.state(&mut ctx).await.get_prev_aum();
    balances = vault.balances(&mut ctx).await;
    xmsg!("*****  token_vault {}, ctokens_vault: {} shares issued: {}, prev_aum: {}",
         balances.token_vault, balances.ctoken_vaults[0], balances.shares_supply,aum);

    let amounts_to_deposit = [3u64,5,9,15,23,35,53,80,120,180,270,405,607,910,1365,2047,3070];
    for (i, to_deposit) in amounts_to_deposit.iter().enumerate() {
        client::deposit(&mut ctx, &vault, &depositor, *to_deposit).await;
```

```
        client::withdraw(&mut ctx, &vault, &depositor, 1, reserve).await;
        balances = vault.balances(&mut ctx).await;
        aum = vault.state(&mut ctx).await.get_prev_aum();
        xmsg!("***** deposit step {} ( {} ), token_vault {}, ctokens_vault: {} shares issued: {}, prev_aum: {}",
                 i, to_deposit, balances.token_vault, balances.ctoken_vaults[0], balances.shares_supply,aum);
    }
    assert_eq!(balances.shares_supply,1);
    assert_gt!(aum,2303);
}
```

Output from the test:

Unset

```
running 1 test

Reserve C7CENCmKwfUqgAffZdLgwjoqBgFbG3pNisuqfxVQr1eA has 100000000 liquidity from 100000000 c tokens
Forward time to increase aum from reserve profits
Start withdrawal of 99999999 shares
new_aum 100000001.6788 prev_aum 100000000.0000 seconds_passed 60
Balances after withdraw

*****  token_vault 1, ctokens_vault: 1 shares issued: 1, prev_aum: 1.678753970290295001
***** deposit step 0 ( 3 ), token_vault 2, ctokens_vault: 1 shares issued: 1, prev_aum: 3.000000016854743064
***** deposit step 1 ( 5 ), token_vault 3, ctokens_vault: 1 shares issued: 1, prev_aum: 4.000000016854743064
***** deposit step 2 ( 9 ), token_vault 6, ctokens_vault: 1 shares issued: 1, prev_aum: 7.000000016854743064
***** deposit step 3 ( 15 ), token_vault 10, ctokens_vault: 1 shares issued: 1, prev_aum: 11.000000016854743064
***** deposit step 4 ( 23 ), token_vault 16, ctokens_vault: 1 shares issued: 1, prev_aum: 17.000000016854743064
***** deposit step 5 ( 35 ), token_vault 25, ctokens_vault: 1 shares issued: 1, prev_aum: 26.000000016854743064
***** deposit step 6 ( 53 ), token_vault 39, ctokens_vault: 1 shares issued: 1, prev_aum: 40.000000016854743064
***** deposit step 7 ( 80 ), token_vault 59, ctokens_vault: 1 shares issued: 1, prev_aum: 60.000000016854743064
***** deposit step 8 ( 120 ), token_vault 89, ctokens_vault: 1 shares issued: 1, prev_aum: 90.000000016854743064
***** deposit step 9 ( 180 ), token_vault 134, ctokens_vault: 1 shares issued: 1, prev_aum: 135.000000016854743064
***** deposit step 10 ( 270 ), token_vault 202, ctokens_vault: 1 shares issued: 1, prev_aum: 203.000000016854743064
***** deposit step 11 ( 405 ), token_vault 303, ctokens_vault: 1 shares issued: 1, prev_aum: 304.000000016854743064
***** deposit step 12 ( 607 ), token_vault 455, ctokens_vault: 1 shares issued: 1, prev_aum: 456.000000016854743064
***** deposit step 13 ( 910 ), token_vault 682, ctokens_vault: 1 shares issued: 1, prev_aum: 683.000000016854743064
***** deposit step 14 ( 1365 ), token_vault 1023, ctokens_vault: 1 shares issued: 1, prev_aum: 1024.000000016854743064
***** deposit step 15 ( 2047 ), token_vault 1535, ctokens_vault: 1 shares issued: 1, prev_aum: 1536.000000016854743064
***** deposit step 16 ( 3070 ), token_vault 2302, ctokens_vault: 1 shares issued: 1, prev_aum: 2303.000000016854743064
```

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.