



Texture 2.0 Security Assessment & Formal Verification Report

December 2024

Prepared for
Texture



Table of Contents

Project Summary.....	4
Project Scope.....	4
Project Overview.....	4
Findings Summary.....	5
Severity Matrix.....	5
Detailed Findings.....	6
Critical-Severity Issues.....	6
C-01 reserve_data.lp_market_price() might be stale when called.....	6
C-02 Vault reward system can lead to locked user funds.....	8
High-Severity Issues.....	12
H-01 Using partially verified price updates is dangerous, as it lowers the threshold of guardians that need to collude to produce a malicious price update.....	12
H-02 Anyone can DOS another user from receiving a rewards.....	13
H-03 Changing rule start date after it has started will result in a loss of rewards for the user.....	15
Medium-Severity Issues.....	17
M-01 Lack of price/expo validation for Pyth price result.....	17
Low-Severity Issues.....	19
L-01 In case of a black swan event or malfunctioning oracle, writing off bad debt can occur before liquidations.....	19
L-02 Conflicting checks in alter texture config.....	21
Informational Issues.....	22
I-01 fees.curator_borrow_fee_rate_bps or fees.curator_performance_fee_rate_bps can't be set to 50% even though the comments say it should be.....	22
I-02 reserve_mode or flash_loans_enabled can be set to arbitrary value.....	23
I-03 Switchboard confidence interval is not utilized.....	25
I-04 The reward rule can be set to an arbitrary value.....	26
Formal Verification.....	27
Verification Notations.....	27
General Assumptions and Simplifications.....	27
Formal Verification Properties.....	29
superlendy/program/src/processor/reserve.rs:.....	29
P-01: Each flash borrow has a subsequent matching flash repay.....	29
P-02: Each flash repay has a preceding matching flash borrow.....	30
P-03: Flash borrow does not interfere with total liquidity.....	32
P-04: Flash borrow and flash repay correctly transfer tokens.....	32
P-05: Integrity of claiming performance fees.....	33
P-06: Integrity of Depositing and withdrawing.....	34



Disclaimer.....	35
About Certora.....	35



Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Texture super-lendy	Texture-Fi Super Lendy	9c563f7	Solana
Texture Vault	Texture-Fi Vault	b5dd945	Solana
Texture price-proxy	Texture-Fi Price Proxy	615aa79	Solana
Texture curvy	Texture-Fi Curvy	c47699a	Solana

Project Overview

This document describes the specification and verification of **SuperLendy Lending protocol** using manual code review and Certora Prover. The work was undertaken from **October 14 , 2024**, to **November 22, 2024**.

The following contract list is included in our scope:

all files under

SuperLendy/

price-proxy/

Curvy/

Vault/

The Certora Prover demonstrated that the implementation of the Solana contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solana contracts in the scope. During the verification process and the manual audit, the Certora team discovered bugs in the Solana contracts code, as listed on the following page.





Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	2	2	2
High	3	3	3
Medium	1	1	1
Low	2	2	2
Informational	4	4	3
Total	12	12	11

Severity Matrix

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



Detailed Findings

Critical-Severity Issues

C-01 reserve_data.lp_market_price() might be stale when called

Severity: **Critical**

Impact: **High**

Likelihood: **High**

Files:
price-proxy/program/src/processor/mod.rs

Category: Logic

Status: Fixed

Description:

When fetching reserve data, there is no validation check for `self.lastUpdate.stale == 1`, even though the staleness check if `unpacked_reserve.is_stale(&clock)?` is properly implemented for liquidity addition/removal and borrowing operations.

JavaScript

```
pub fn check_price_feed(
    &self,
    market_price_feed: &'a AccountInfo<'b>,
    pool: &'a AccountInfo<'b>,
) -> LendyResult<()> {
    let price_feed_data = market_price_feed.data.borrow();
    let unpacked_price_feed = PriceFeed::try_from_bytes(&price_feed_data)?;
    let feed_quote_symbol = unpacked_price_feed.quote_symbol();
```

**Impact:**

Stale or manipulated price data could be returned and used in calculations, even in cases where the standard staleness check appears to pass. This creates a risk of using outdated or incorrect price information in critical financial operations.

Recommendation:

Add a staleness validation check by implementing `reserve_data.last_update.stale == 1` verification when fetching reserve data

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



C-02 Vault reward system can lead to locked user funds

Severity: **Critical**

Impact: **High**

Likelihood: **High**

Files:
vault/program/src/processor/position.rs

Category: Logic

Status: Fixed

Description:

When attempting to withdraw from the vault, the function attempts to ensure that the user will withdraw all their rewards by doing a last-minute accrual:

Python

```
let touched_rewards_records = unpacked_position.rewards.accrue_rewards(  
    unpacked_position.issued_vlp_amount()?,  
    &unpacked_vault.reward_rules,  
    clock.slot,  
)?;
```

When we take a look at program/src/state/rewards.rs::accrue_rewards:

Python

```
pub fn accrue_rewards(  
    &mut self,
```



```
amount: Decimal,
reward_rules: &RewardRules,
current_slot: Slot,
) -> VaultResult<HashSet<usize>> {
    let mut touched_reward_records = HashSet::new();

    for rule in reward_rules.rules {

        // 🦋 check whether if any of the existing rewards matches the new vault rewards
        match self.find_reward(&rule.reward_mint) {
            // 🦋 if they do not
            None => {
                // 🦋 attempt to find a free or unused reward
                match self.find_unused_reward() {
                    // 🦋 if you can't find
                    None => {
                        msg!(
                            "warning: no free space for rewards for rule {} reward_mint
                            {}\"",
                            String::from_utf8_lossy(&rule.name),
                            rule.reward_mint
                        );
                        // error here
                        return Err(VaultError::Internal("No free space for
                            rewards".into()));
                    }
                }
            }
        }
    }
}
```

We can notice that if there are not enough free reward spaces, the transaction will revert because of the “No free space for rewards” error.

This can occur when, for example, there have been 8 rewards (the maximum), some of them expired and the user has not claimed them yet, but the admin has created more rewards that are ‘yet-to-be-initialized’ for that user account.

The only way to clear the reward slots is the rewards are claimed:



Python

```
pub fn claim(&mut self, amount: u64) -> VaultResult<()> {
    if amount > self.accrued_amount {
        msg!(
            "amount to claim {} can not be greater accrued amount {}",
            amount,
            self.accrued_amount
        );
        return Err(VaultError::OperationCannotBePerformed);
    }

    self.accrued_amount -= amount;


    if self.accrued_amount == 0 {
>>         self.reward_mint = Pubkey::default(); // Mark record as unused
>>         self.accrued_slot = 0;
    }

    Ok(())
}
```

However, in `vault/program/src/processor/rewards.rs::claim_reward` we can observe that before claiming the rewards, it is calling `accrue_rewards` as well, which would lead to the same error.

Impact:

To summarize, users get stuck in this loop:

1. Can't withdraw because they need to claim rewards
2. Can't claim rewards because they need free slots
3. Can't free slots because they need to claim rewards
4.  repeat forever, cause there's no way to break the deadlock for a specific user unless the admin gives up on the reward system.

The root cause is that both `withdraw` and `claim_reward` try to `accrue` rewards first, but `accrual` can fail if slots are full.

Recommendation:



1. Remove the reward accrual from claim_reward (simplest fix)
2. Or add a force-claim option that skips accrual for emergencies
3. Most robust: Allow claiming specific rewards without accruing others

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



High-Severity Issues

H-01 Using partially verified price updates is dangerous, as it lowers the threshold of guardians that need to collude to produce a malicious price update

Severity: **High**

Impact: **High**

Likelihood: **low**

Files:
price-proxy/program/src/processo
r/mod.rs

Category: Logic

Status: Fixed

Description:

When we look at how the Pyth price is consumed, we can see the verification level is partial.

Unset

```
let num_signatures = 5;
let pyth_price = price_update
    .get_price_no_older_than_with_custom_verification_level(
        &Clock::get().expect("clock"),
        maximum_age_sec,
        &price_update.price_message.feed_id,
        VerificationLevel::Partial { num_signatures },
    )
>>>
```

However, in the SDK's [docs](#), we can observe the following warning:

Warning

Lowering the verification level from **Full** to **Partial** increases the risk of using a malicious price update. Please read the documentation for **VerificationLevel** for more information.

**Impact:**

By default, wormhole recommends checking 2/3 of guardian signatures, which means its 13/18. Partial verification allows you to check fewer signatures than 13. If you check N signatures and any N guardians among all get compromised, they'd be able to submit arbitrary prices

Recommendation:

Use the full verification level.

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.

H-02 Anyone can DOS another user from receiving a rewards**Severity: High****Impact: Medium****Likelihood: High**

Files:
superlendy/program/src/state/position.rs

Status: Fixed

Description:

There is a DOS vulnerability in SuperLendy where users can prevent others from receiving rewards by forcing position refreshes. The issue stems from reward calculations rounding down combined with the position refresh mechanism.

Exploit scenario:

- When a user's position is refreshed, it recalculates and accrues rewards



- Due to rounding down in reward calculations, frequent refreshes can result in 0 rewards
- An attacker can repeatedly refresh another user's position, causing their rewards to consistently round down to 0

For example:

- Alice has a position earning 1 token per 100 slots
- Bob (attacker) keeps calling refresh_position on Alice's position
- If Bob refreshes every 99 slots, Alice's rewards always round down to 0
- Since position refresh is permissionless (anyone can call it), Alice can't prevent this

Impact:

This is problematic because:

- Anyone can DOS another user's reward accrual
- Attack only costs transaction fees
- Hard for users to detect they're being targeted
- Could be used to grief specific users or manipulate protocol behavior

Recommendation:

Probably the simplest fix is to not update the accrual slot if the rewards are 0.

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



H-03 Changing rule start date after it has started will result in a loss of rewards for the user

Severity: **High**

Impact: **Medium**

Likelihood: **High**

Files:
superlendy/program/src/state/position.rs

Status: Fixed

Description:

The vulnerability exists in the reward accrual logic where in the `accrue()` function in `reward.rs`, the reward calculation uses:

Unset

```
let slots_since_last_accrual = current_slot - self.accrued_slot;  
let rule_age = current_slot - rule.start_slot;  
let slots_to_accrue = min(slots_since_last_accrual, rule_age);
```

The vulnerability arises because:

- The reward calculation takes the minimum between slots since the last accrual and the rule age
- If an admin changes the rule's `start_slot` to a later date after users have started accruing rewards, the `rule_age` will become smaller
- This means users will lose rewards they should have earned during the period between their last accrual and the new start date

Consider the following scenario:

- User starts accruing rewards on Nov 2 (slot X)



- Admin changes start date to Nov 6 (slot Y where $Y > X$)
- When a user performs any action on Nov 7:
 - $\text{slots_since_last_accrual} = \text{current_slot} - X$
 - $\text{rule_age} = \text{current_slot} - Y$ (smaller than it should be)
 - $\text{slots_to_accrue} = \min(\text{current_slot} - X, \text{current_slot} - Y) = \text{current_slot} - Y$
 - All rewards between X and Y are lost

Impact

The vulnerability breaks the core principle that earned rewards should not depend on when users claim them

Recommendation:

Add a check in `set_reward_rules()` that prevents changing the `start_slot` of an existing rule to a later date

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



Medium-Severity Issues

M-01 Lack of price/expo validation for Pyth price result

Severity: **Medium**

Impact: **High**

Likelihood: **Low**

Files:
price-proxy/program/src/proce
ssor/mod.rs

Category: TBD

Status: Fixed

Description: If we look at the [EVM SDK](#) implementation of the Pyth Utils, we can notice that they are checking whether the price > 0 or the expo is positive.

However, these checks are missing in the Solana one and we must implement them independently.

Note, this was confirmed with the Pyth team:

> We agree that it was not applied for Solana as we missed that. Currently, we don't have negative prices coming for any of the feeds, but it can be in the future or if the Oracle malfunctions.

> We feel it's better to check the expo for specific use cases. It can give positive expos if they subscribe to the wrong feeds of different asset types.

Impact:

In case of oracle malfunction or subscribing to wrong feeds,

Recommendation:

Apply the following checks:



Python

```
if (price < 0 || expo > 0 || expo < -255)
```

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



Low-Severity Issues

L-01 In case of a black swan event or malfunctioning oracle, writing off bad debt can occur before liquidations

Severity: **Low**

Impact: **Low**

Likelihood: **Low**

Files:
superlendy/program/src/processor/position.rs

Status: Fixed

Description:

#1 The oracle is refreshed and the price feed returns a malfunctioning price or 0 (in case of a black swan event)

Java

```
```rust
pub fn refresh_reserve(&self) -> LendyResult<()> {
 msg!("refresh_reserve ix");

 >> unpacked_reserve
 .liquidity
 .set_market_price(unpacked_price_feed.try_price())?;
 ...
```
```

#2 When the position is refreshed, that would set the deposited_value of the position to 0.



Java

```
pub fn refresh_position(&self, deposit_count: usize, borrow_count: usize) ->
LendyResult<()> {
    ...
>>     let market_value = deposit_reserve
        .lp_exchange_rate()?
        .decimal_lp_to_liquidity(collateral.deposited_amount.into())?
        .checked_mul(deposit_reserve.liquidity.market_price())?
        .checked_div(Decimal::from(decimals))?;

>>     deposited_value = deposited_value.checked_add(market_value)?;
```

#3 Which would then pass the check in `write_off_bad_debt` as if the liquidation has happened, but it didn't.

Java

```
pub fn write_off_bad_debt(&self, amount: u64) -> LendyResult<()> {
    ...
>>     if unpacked_position.deposited_value()? != Decimal::ZERO {
        msg!("Position has {} deposited value. Liquidate it first and then try to write
off bad debt again.", unpacked_position.deposited_value()?);
        return Err(SuperLendyError::OperationCanNotBePerformed);
    }
```

Impact:

This would allow for bad debt to be written off before liquidations, therefore the reserve accounting will be invalid/flawed.

Recommendation:

You can either revert when `0` is returned from the oracle or only write off bad debt when the amount is 0.

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



L-02 Conflicting checks in alter texture config

| | | |
|---|--------------------|------------------------|
| Severity: Low | Impact: Low | Likelihood: Low |
| Files:
superlendy/program/src/proces
sor/mod.rs | | Status: Fixed |

Description:

One of the checks in params.validate is:

Python

```
if params.performance_fee_rate_bps >= 5000 {  
    msg!("performance_fee_rate_bps must be in range [0, 50] %");  
    return Err(SuperLendyError::InvalidConfig);  
}
```

But then, downstream there is another:

Python

```
if self.performance_fee_rate_bps > 4000 {  
    msg!("Performance fee rate must be in range [0, 40] %");  
    return Err(SuperLendyError::InvalidConfig);  
}
```

Impact:

Even though the first validation might pass, the second would fail.

Recommendation:

Stick with only 1 of the checks, preferably in params.validate.

Customer's response: Will fix.



Fix review: The vulnerability appears properly remediated.

Informational Issues

I-01 fees.curator_borrow_fee_rate_bps or fees.curator_performance_fee_rate_bps can't be set to 50% even though the comments say it should be

Files:
superlendy/program/src/proces
sor/reserve.rs

Status: Confirmed

Description:

Java

```
>> if self.fees.curator_borrow_fee_rate_bps >= 5000 {  
    msg!("curator_borrow_fee must be in range [0, 50) %");  
    return Err(SuperLendyError::InvalidConfig);  
}  
  
>> if self.fees.curator_performance_fee_rate_bps >= 5000 {  
    msg!("curator_performance_fee_rate_bps must be in range [0, 50) %");  
    return Err(SuperLendyError::InvalidConfig);  
}
```

Recommendation:

Java

```
- if self.fees.curator_borrow_fee_rate_bps >= 5000 {  
+ if self.fees.curator_borrow_fee_rate_bps > 5000 {  
    msg!("curator_borrow_fee must be in range [0, 50) %");  
}
```



```

        return Err(SuperLendyError::InvalidConfig);
    }

-     if self.fees.curator_performance_fee_rate_bps >= 5000 {
+     if self.fees.curator_performance_fee_rate_bps > 5000 {
        msg!("curator_performance_fee_rate_bps must be in range [0, 50) %");
        return Err(SuperLendyError::InvalidConfig);
    }

```

Customer's response: Confirmed, will be fixed in future protocol upgrades.

I-02 reserve_mode or flash_loans_enabled can be set to arbitrary value

Files:
superlendy/program/src/processor/reserve.rs

Status: Fixed

Description:

If we look at `alter_reserve` we can notice that `mode` and `flash_loans_enabled` are uint's.

```

Unset
pub fn alter_reserve(
    &self,
    proposed_config: ReserveConfig,
    >     mode: u8,
    >     flash_loans_enabled: u8,
) -> LendyResult<()> {
    msg!("alter_reserve ix: {:?}", proposed_config);
    ...
    unpacked_reserve.mode = mode;

```




```
    unpacked_reserve.flash_loans_enabled = flash_loans_enabled;  
}
```

But at the same time, the only possible modes are 0,1,2 and for flash_loans_enabled it is 0 and 1.

Unset

```
pub const RESERVE_MODE_NORMAL: u8 = 0;  
pub const RESERVE_MODE_BORROW_DISABLED: u8 = 1;  
pub const RESERVE_MODE_RETAIN_LIQUIDITY: u8 = 2;
```

However, right now nothing prevents setting a wrong configuration through alter reserve. For example, the `flash_borrow` function checks whether `flash_loans_enabled == 0`, and based on that it assumes they are disabled.

Python

```
if unpacked_reserve.flash_loans_enabled == 0 {  
    msg!("Flash borrow disabled for that reserve.");  
    return Err(SuperLendyError::OperationCanNotBePerformed);  
}
```

However, given that it can be set to any value (not only 0,1) with a wrong configuration, the function might also assume they are enabled.

Recommendation:

Ensure the modes can be only 0,1,2 and flash_loans_enabled can only be 0 and 1.

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



I-03 Switchboard confidence interval is not utilized

Files:
superlendy/program/src/proces
sor/reserve.rs

Status: Fixed

Description:

Currently, when using the switchboard feed, the code does not implement the confidence interval from the feed.

Python

```
data_feed.check_staleness(Clock::get().unwrap().unix_timestamp, maximum_age_sec as  
i64).expect("price is stale");
```

Recommendation:

Consume the price feed as per switchboard's [official](#) example

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



I-04 The reward rule can be set to an arbitrary value

Files:
superlendy/program/src/processor/reserve.rs

Status: Fixed

Description:

The superlendy rewards have a reason (0 for borrowing and 1 for liquidity). However, when setting the rules there are no checks that

Unset

```
pub fn set_reward_rules(&self, mut rules: RewardRules) -> LendyResult<()> {  
    ...  
    for rule in rules.rules.iter_mut() {  
        if rule.start_slot == 0 {  
            rule.start_slot = clock.slot;  
        }  
    }  
  
    >>    unpacked_reserve.reward_rules = rules;  
}
```

Recommendation:

Ensure that the reward rules are either 0 or 1.

Customer's response: Will fix.

Fix review: The vulnerability appears properly remediated.



Formal Verification

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 counter-example exists that violates one of the assertions of the rule. |

General Assumptions and Simplifications

1. We use the Solana platform tools v1.41
2. Use of mocks
 - a. We use mocks for the SPL token program. These mocks have been formally verified. The mocks are available at <https://github.com/Certora/solana-cvt>.
 - b. Inside the loops in `flash_borrow` and `flash_repay`, we substitute `Instruction` with a struct `CvtInstruction` which limits the number of accounts to two and the size of `data` to 256 bytes. Limiting the number of accounts does not interfere with the correctness of the verification since only two accounts are used in the loops inside of `flash_borrow` and `flash_repay`. Limiting the size of `data` to 256 bytes does not interfere with the correctness of the verification since we mock unpacking `SuperLendyInstructions` with a function that returns a nondeterministic `SuperLendInstruction` by ignoring the data (see d.).



- c. We mock the calls to `load_instruction_at_checked` inside of the loops inside `flash_borrow` and `flash_repay` with a function that returns a nondeterministic `CvtInstruction` or an error.
 - d. We mock the unpacking of `SuperLendyInstructions` in the loops inside `flash_borrow` and `flash_repay` with calls to a function that returns either a nondeterministic `SuperLendyInstruction` or an unpacking error.
 - e. Mock for Decimals: We use a mock implementation of the Decimals library to simplify arithmetic operations. This mock library retains the core functionality of the original but employs basic arithmetic operations, omitting the rounding computation strategies for simplicity.
3. Loop iterations: Any loop was unrolled at most 3 times (iterations)



Formal Verification Properties

superlendy/program/src/processor/reserve.rs:

Contract Properties

P-01: Each flash borrow has a subsequent matching flash repay

Status: Verified

| Rule Name | Status | Description | Links |
|---|----------|--|-----------------------------|
| rule_flash_borrow_loop_increases_index | Verified | This rule verifies that one iteration of the loop in the <code>flash_borrow</code> function increases the index of the instruction that is being considered. This implies that the loop considers only instructions that will be executed after the borrowing. | Rule report |
| rule_flash_borrow_loop_body_if_break_then_last_instruction_is_repay | Verified | This rule verifies that if one iteration of the <code>flash_borrow</code> loop results in a break, then the last loaded instruction must be a <code>FlashRepay</code> , which repays the reserve from which the borrow was made, and the amount is exactly matched. | Rule report |
| rule_flash_borrow_loop_body_if_continue_the_n_continue_precondition | Verified | This rule verifies that if one iteration of the <code>flash_borrow</code> loop results in a continue, then one of the following holds:

1) the instruction is not for the program that is being executed;

2) the last loaded instruction is a <code>FlashRepay</code> for another reserve;

3) the last loaded instruction is a | Rule report |



| | | | |
|--|----------|---|-----------------------------|
| | | FlashBorrow for another reserve;

4) the last loaded instruction is any other instruction that is not a FlashRepay or a FlashBorrow. | |
| rule_flash_borrow_loop_body_if_error_then_error_precondition | Verified | <p>This rule verifies that if one iteration of the <code>flash_borrow</code> loop returns an error (<code>Err</code>), then one of the following holds:</p> <ol style="list-style-type: none"> 1) the last loaded instruction has not been loaded correctly; 2) the last loaded instruction data does not represent a valid <code>SuperLendyInstruction</code>; 3) the last loaded instruction is a <code>FlashRepay</code> for the same reserve, but the repay amount does not match the borrow amount; 4) the last loaded instruction is <code>FlashBorrow</code> for the same reserve. | Rule report |

P-02: Each flash repay has a preceding matching flash borrow

Status: Verified

| Rule Name | Status | Description | Links |
|---------------------------------------|----------|---|-----------------------------|
| rule_flash_repay_loop_decreases_index | Verified | <p>This rule verifies that one iteration of the loop in the <code>flash_repay</code> function decreases the index of the instruction that is being considered. This implies that the loop considers only instructions that will be executed before the repay.</p> | Rule report |



| | | | |
|---|----------|---|-----------------------------|
| rule_flash_repay_loop_body_if_break_then_last_instruction_is_borrow | Verified | This rule verifies that if one iteration of the <code>flash_repay</code> loop results in a break, then the last loaded instruction must be a <code>FlashBorrow</code> , which borrows from the reserve to which the repayment is made, and the amount must be an exact match. | Rule report |
| rule_flash_repay_loop_body_if_continue_then_continue_precondition | Verified | <p>This rule verifies that if one iteration of the <code>flash_repay</code> loop results in a continue, then one of the following holds:</p> <ol style="list-style-type: none"> 1) the instruction is not for the program that is being executed; 2) the last loaded instruction is a <code>FlashRepay</code> for another reserve; 3) the last loaded instruction is a <code>FlashBorrow</code> for another reserve; 4) the last loaded instruction is any other instruction that is not a <code>FlashRepay</code> or a <code>FlashBorrow</code>. | Rule report |
| rule_flash_repay_loop_body_if_error_then_error_precondition | Verified | <p>This rule verifies that if one iteration of the <code>flash_borrow</code> loop returns an error (<code>Err</code>), then one of the following holds:</p> <ol style="list-style-type: none"> 1) the last loaded instruction has not been loaded correctly; 2) the last loaded instruction data does not represent a valid <code>SuperLendyInstruction</code>; 3) the last loaded instruction is a <code>FlashRepay</code> for the same reserve; 4) the last loaded instruction is <code>FlashBorrow</code> for the same reserve, but the borrow amount does not match the repay amount. | Rule report |



P-03: Flash borrow does not interfere with total liquidity

Status: Verified

| Rule Name | Status | Description | Links |
|---|----------|--|-----------------------------|
| rule_lend_does_not_interfere_in_total_liquidity | Verified | This rule verifies that the arbitrary sequence of instructions executed between a <code>flash_borrow</code> and a <code>flash_repay</code> is the only factor capable of affecting total liquidity. Namely, a properly matched borrow-repay pair does not alter the overall liquidity. | Rule report |

P-04: Flash borrow and flash repay correctly transfer tokens

Status: Verified

| Rule Name | Status | Description | Links |
|------------------------------------|----------|--|-----------------------------|
| rule_flash_borrow_transfers_tokens | Verified | This rule verifies that <code>flash_borrow</code> increases the token amount in the destination wallet by the borrowed amount and that it decreases the token amount in the liquidity supply by the same amount. | Rule report |
| rule_flash_repay_transfers_tokens | Verified | This rule verifies that <code>flash_repay</code> increases the token amount in the liquidity supply by the repaid amount and that it decreases the token amount in the source wallet by the same amount. | Rule report |



P-05: Integrity of claiming performance fees

Status: Verified

| Rule Name | Status | Description | Links |
|---|----------|--|-----------------------------|
| rule_claim_curator_performance_fees_integrity | Verified | <p>The correct amount of fees transferred from The protocol was received by the fee_receiver.</p> <p>Fees transferred out of the protocol.</p> <p>Fees transferred to the right address.</p> <p>Claiming fees do not change the collateral available amount.</p> <p>Claiming fees do not change the lp total supply.</p> <p>Fees left in the reserves are less than one wad.</p> | Rule report |
| rule_claim_texture_performance_fees_integrity | Verified | <p>The correct amount of fees transferred from The protocol was received by the fee_receiver.</p> <p>Fees transferred out of the protocol.</p> <p>Fees transferred to the right address.</p> <p>Claiming fees do not change the collateral available amount.</p> <p>Claiming fees do not change the lp total supply.</p> <p>Fees left in the reserves are less than one wad.</p> | Rule report |



P-06: Integrity of Depositing and withdrawing

Status: Verified

| Rule Name | Status | Description | Links |
|--------------------------------------|----------|---|-----------------------------|
| rule_integrity_of_deposit_liquidity | Verified | Verifies that after a successful call for 'deposit_liquidity', the available amount, depositor's balance and the reserve's balance are updated correctly. | Rule report |
| rule_integrity_of_withdraw_liquidity | Verified | Verifies that after a successful call for 'withdraw_liquidity', the collateral's lp_total_supply and the withdrawer's balance are updated correctly. | Rule report |



Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

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.