certora

Formal Verification

FLUID Jupiter Lend

Jupiter Lend

September - November 2025

Prepared for Jupiter Lend/Fluid

Table of content

certora

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
Fluid Solana	/Instadapp/fluid-contracts-solana	7069f55	Solana

Project Overview

This document describes the specification and verification of Jupiter Lend protocol on Solana using Certora prover. The work was undertaken from September 15th 2025 to December 1st 2025.

All files in the repository are considered in scope, but the main focus for formal verification was on the following programs:

- `Liquidity`
- `Vaults`
- `Lending`

The Certora Prover demonstrated that the implementation of the Solana contracts above is correct with respect to the formal rules designed and written by the Certora team.

The Certora team additionally performed a manual audit that covered the `FlashLoan`, `Oracle` and `LendingRewardRateModel` programs as these were not covered by the formal verification rules. We include the findings of the partial manual review below.

Protocol Overview

Fluid is a capital-efficient DeFi protocol originally built in the EVM ecosystem. Its architecture unifies lending and vaults into a common liquidity layer, enabling high capital reuse, dynamic risk control, and modular composability. Fluid has extended its core design to Solana through a partnership with Jupiter, launching Jupiter Lend (powered by Fluid).

Core Architecture & Design Principles

The core of the protocol is the **Liquidity program.** It tracks total deposits and withdrawals per asset, handles interest accrual, and provides the base liquidity for borrowing and lending.

The liquidity program provides the foundation for multiple sub-protocols, including:

- **Lending program:** Users supply assets to earn yield. Deposits are tracked as fTokens representing a claim on the pooled liquidity. Interest accrual is handled at the module level, leveraging the shared liquidity for efficiency.
- **Vaults program:** Allows over-collateralized borrowing. Collateral and debt are mapped into ticks, discrete risk bags aggregating many user positions. Updates to interest, utilization, and collateral health are performed per tick rather than per individual position.
  Liquidations also operate at the tick level: when a tick's aggregate health falls below TVL thresholds, the entire tick is liquidated and the effects are distributed proportionally across all positions in that tick. This approach reduces computation and storage overhead and enables batch liquidation of large numbers of positions.
- **Flashloan program:** Provides permissionless, atomic flashloans of liquidity-layer assets. Users can borrow any amount available in the liquidity pool within a single transaction, provided the borrowed amount plus fees is returned before transaction completion.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	0	0	0
Medium	11	2	1
Low	3	1	1
Informational	1	1	1
Total	15	0	0

Severity Matrix

Impact	Critical	Medium	Medium/High	High	Critical
	High	Low/Medium	Medium	Medium/High	High
	Medium	Low	Low/Medium	Medium	Medium/High
	Low	Informational	Low	Low/Medium	Medium
		Rare	Unlikely	Likely	Very Likely
		Likelihood			

Detailed Findings

ID	Title	Severity	Status
M-01	Redstone oracle integration uses millisecond timestamps without conversion lead to reversion	Medium	Fixed
M-02	Rounding issue can DoS flashloan transactions when the flashloan fee is set to the max of 1%	Medium	Fixed
M-03	Liquidity and Vaults borrow_exchange_price calculation rounds down	Medium	Fixed
M-04	Borrow_fee rounds down in vaults	Medium	Fixed
M-05	Liquidity clients use input amount instead of calculated value	Medium	Fixed
M-06	Utilization scaling division done over rounded values	Medium	Fixed
M-07	Incorrect rounding in liquidate debt	Medium	Fixed
M-08	Incorrect rounding in calc_revenue	Medium	Fixed

M-09	Incorrect rounding direction in unscale debt amount	Medium	Fixed
M-10	Incorrect rounding direction col_per_debt	Medium	Fixed
M-11	Inconsistent vault updates when actual_debt_amt > debt_amount	Medium	Fixed
L-01	Oracle source multiplier configuration is not applied to rate calculation	Low	Fixed
L-02	Pyth oracle does not allow future timestamps which can result in DoS when the solana network's clock drifts away even by just a few seconds	Low	Fixed
L-03	Mint and metadata accounts are not closed, permanently locking rent funds	Low	Acknowledged
I-01	Oracle exponent exceeding rate decimals causes underflow	Informational	Fixed

Medium Severity Issues

M-01 Redstone oracle integration uses millisecond timestamps without conversion lead to reversion

Severity: **Medium**	Impact: **Medium**	Likelihood: **Likely**
Files: redstone.rs#L33-L43	Status: Fixed	

Description:

RedStone's oracle timestamps are provided **in milliseconds**. However, the integration extracts the timestamp and passes it directly to `verify_publish_time()` without converting it to seconds. The `verify_publish_time()` function expects a timestamp in seconds, as it compares against `MAX_AGE_OPERATE` (600 seconds = 10 minutes) and `MAX_AGE_LIQUIDATE` (7200 seconds = 2 hours).

Currently, the code computes:

```rust
let publish_time = if redstone_feed.write_timestamp.is_some() {
    // return minimum of timestamp and write_timestamp

redstone_feed.timestamp.min(redstone_feed.write_timestamp.unwrap()).c
ast()?
} else {
    return err!(ErrorCodes::TimestampExpected);
};
```

Since the timestamp is inflated by 1000x, when `verify_publish_time()` calculates `current_time.safe_sub(publish_time)?`, the result will underflow. This means getting the price from RedStone oracle will always revert.

Recommendations:

When calling the verify function, divide by **1000**:

```rust
verify_publish_time(publish_time / 1000, is_liquidate)?;
```

Customer's response: Fixed in 5ac8d11

Fix Review: Fix confirmed

M-02 Rounding issue can DoS flashloan transactions when the flashloan fee is set to the max of 1%

Severity: **Medium**	Impact: **Medium**	Likelihood: **Unlikely**
Files: user.rs#L86-L94	Status: Fixed	

Description:

When using the max `FLASHLOAN_FEE_MAX` which is currently 1%. If the flash-borrow amount in tokens is not completely divisible by `100` (i.e. last 2 digits of the amount are not `00`), this ceils up the fee amount and this results in the `MAX_INPUT_AMOUNT_EXCESS` liquidity program validation failing with `TransferAmountOutOfBounds`

It's quite likely that the last 2 digits are almost never 0, given generally low decimals for tokens on Solana combined with auto-calculated amounts for arbitrage. Already successful flashloan txs confirm this:
https://solscan.io/tx/2ir19po8BgAPjrY5vguMG6M4ceNnxGMUzY1TAx9R69abDXQy2XkjRMdTSMNvowxBBwsRo35W4rGicqmeCGwdK3pJ

During the `flashloan_payback` operation, we do 3 main things:

1. pre_operate_with_signer - this is called on the liquidity program and saves a snapshot of the balance
2. transfer_spl_tokens - this transfers the owed tokens (this includes fees + borrowed amount)
3. accounts.operate_with_signer - this calls the operate function on the liquidity program which includes the processing logic and this is where the `TransferAmountOutOfBounds` check fail occurs

In the following snippet this is done and we also compute amount_with_fee there, some things should be noted: `active_flashloan_amount = amount` this is validated by `validate_flashloan_payback`

```
Rust
pub fn flashloan_payback(ctx: Context<Flashloan>, amount: u64) -> Result<()> {

        let active_flashloan_amount = ctx.accounts.flashloan_admin.active_flashloan_amount;

@>>     validate_flashloan_payback(active_flashloan_amount, amount)?;

        if ctx.accounts.flashloan_admin.is_paused() {

           return Err(ErrorCodes::FlashloanPaused.into());

        }

@>>     let amount_with_fee: u64 = ctx

            .accounts

            .flashloan_admin

            .get_expected_payback_amount(amount)?;


        let bump = ctx.accounts.flashloan_admin.bump;

        let signer_seeds: &[&[&[u8]]] = &[&[FLASHLOAN_ADMIN_SEED, &[bump]]];

        let accounts = ctx.accounts.get_payback_accounts();

        accounts.pre_operate_with_signer(

            PreOperateInstructionParams {

                mint: ctx.accounts.mint.key(),

            },

            signer_seeds,

        )?;
```

```
@>>     transfer_spl_tokens(TokenTransferParams {

            source: ctx.accounts.signer_borrow_token_account.to_account_info(),

            destination: ctx.accounts.vault.to_account_info(),

            authority: ctx.accounts.signer.to_account_info(),

            amount: amount_with_fee,

            token_program: ctx.accounts.token_program.to_account_info(),

            signer_seeds: None,

            mint: ctx.accounts.mint.clone(),

        })?;


@>>      accounts.operate_with_signer(

            OperateInstructionParams {

                supply_amount: 0,

                borrow_amount: active_flashloan_amount.cast::<i128>()?.safe_mul(-1)?, //
payback amount is negative

                withdraw_to: ctx.accounts.liquidity.key(), // default withdraw_to will be
liquidity PDA itself

                borrow_to: ctx.accounts.liquidity.key(), // default borrow_to will be
liquidity PDA itself

            },

            signer_seeds,

        )?;

.....
```

Where in `get_expected_payback_amount()` we round up:

```
Rust
     fn calculate_flashloan_fee(&self, amount: u64) -> Result<u64> {
        let flashloan_fee: u128 = self.flashloan_fee.cast()?;
        let flashloan_fee_amount: u128 = amount
            .cast::<u128>()?
            .safe_mul(flashloan_fee)?
@>>          .safe_div_ceil(FOUR_DECIMALS)?; // round up for flashloan fee

        Ok(flashloan_fee_amount.cast()?)
    }

    pub fn get_expected_payback_amount(&self, amount: u64) -> Result<u64> {
@>>     let flashloan_fee_amount = self.calculate_flashloan_fee(amount)?;
        let expected_payback_amount = amount.safe_add(flashloan_fee_amount)?;

        Ok(expected_payback_amount)
    }
```

And then this value is calculated with `get_expected_payback_amount` which is used for the `transfer_spl_tokens` operation above.

Then in the liquidity program `users.rs` and in operate function, the bad check is in the `net_amount_in > operate_amount_in` if-case

Rust
```rust
pub fn operate() -> Result<(u64, u64)> {

.....

    // operateAmountIn: deposit + payback

    let operate_amount_in: u128 =
(supply_amount).max(0).safe_add((-borrow_amount).max(0))?.cast()?;

    let mut token_reserve = ctx.accounts.token_reserve.load_mut()?;

    let mint = ctx.accounts.mint.clone();

    if operate_amount_in > 0 {

.......

        let last_recorded_balance: u128 = token_reserve.get_interacting_balance()?;

        let final_recorded_balance: u128 =

            balance_of(&ctx.accounts.vault.to_account_info())?.cast()?;

        // final balance - initial balance

        let net_amount_in: u128 = final_recorded_balance.safe_sub(last_recorded_balance)?;


@>>  if net_amount_in < operate_amount_in

            || net_amount_in > operate_amount_in

                    .safe_mul(FOUR_DECIMALS.safe_add(MAX_INPUT_AMOUNT_EXCESS)?)?

                    .safe_div(FOUR_DECIMALS)?

        {

            return Err(ErrorCodes::TransferAmountOutOfBounds.into());

        }

    }
```

The `net_amount_in` represents the amount we transferred, which includes the `borrowed amount + the fee`. The `operate_amount_in` equals exactly the originally borrowed amount, as when we call `accounts.operate_with_signer(` `supply_amount` is `0`.)

This means that there is the following possibility:

1. `1,000,001` tokens are borrowed (not perfectly divisible by `100`, which is 1% in bps) (or any number, doesn't matter token size, its just needed to trigger ceiling)
2. The fee in `calculate_flashloan_fee` rounds this up. Since its the max of `100`, then the calculation is as follows: `1,000,001 * 100 = 100,000,100 / 10,000 = 10,000.01 => 10,001` (round up) Or `1,000,001 + 10,001 = 1,010,002 total payback to be transferred, amount + fee`
3. Then in `operate()` call during the payback we have `net_amount_in > (operate_amount_in * (1e4 + 100) ) / 1e4`
4. Which resolves to `1,010,002 > (1,000,001 * 10,100) / 10,000` and then to `1,010,002 > 1,010,001`, thus its `true`
5. This results in `ErrorCodes::TransferAmountOutOfBounds` and the operation fails

Recommendations:
This could be fixed by ceiling up in the if-check in `operate()` function.

Customer's response: Fixed in PR via commit 1304306

Fix Review: Fix confirmed.

M-03 Liquidity and Vaults borrow_exchange_price calculation rounds down		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: token_reserve.rs vault_state.rs	Status: Fixed	Violated Rules: update_price_fee_period_lemma

Description:
In the liquidity and vaults program, the borrow exchange price is used to calculate how much borrowers owe to the protocol, while the supply exchange price is used to calculate how much the protocol owes to users.
For price updates, rounding direction should always be done in favor of the protocol, which means for borrow_exchange_price, it should round up.

When updating prices, the new borrow_exchange_price is calculated with the following formula (in token_reserve.rs):

```rust
borrow_exchange_price = borrow_exchange_price.safe_add(
    borrow_exchange_price
        .safe_mul(borrow_rate)?
        .safe_mul(seconds_since_last_update)?
        .safe_div(SECONDS_PER_YEAR.safe_mul(FOUR_DECIMALS)?)?,
)?;
```

As can be seen, the price is rounding down.

In some edge cases, this can cause the borrow price to increase at a lower rate than the supply price, potentially decreasing solvency.
The effects of this rounding are minimal, but it is still recommended to round in the correct direction.

Recommendations:
Replace `safe_div` with `safe_div_ceil` for this calculation to enforce rounding up.

Customer's response: Fixed in PR.

Fix Review: Fix confirmed.

M-04 Borrow_fee rounds down in vaults		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: position.rs	Status: Fixed	

Description:
When borrowing from the vault, a user can be charged a borrow_fee.
The fee is added to the borrow_amount and registered as the user's debt.

The fee amount is calculated by multiplication of the borrow_fee BPT and borrow_amount:

```rust
let borrow_amount_with_fee: u128 = borrow_amount.safe_add(
    borrow_amount
        .safe_mul(exchange_prices.borrow_fee.cast()?)?
        .safe_div(FOUR_DECIMALS)?,
)?;
```

The division by FOUR_DECIMALS is rounding down, making it possible that the user is paying a slightly smaller fee. This could incentivise users to do multiple small borrows to pay less fee, compared to one borrow.

Though the impact of the rounding is small, it is recommended to round in favor of the protocol, which in this case is rounding up.

Recommendations:
Replace `safe_div` with `safe_div_ceil` for this calculation to enforce rounding up.

Customer's response: Fixed in PR.

Fix Review: Fix confirmed.

M-05 Liquidity clients use input amount instead of calculated value		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: Vaults: users.rs Lending: deposit.rs	Status: Fixed	

Description:
When interacting with liquidity, the client protocols currently use the amount sent to `liquidity.operate()` as the basis for their calculations.

Due to rounding in liquidity, the value registered in liquidity can be slightly different. It is recommended to use the actual value that is registered in liquidity as a basis for further calculations

Example:
When depositing supply to liquidity, the `usersupplyPosition.amount` and `token_reserve.total_supply` are both increased by `amount / supply_exchange_price` which is rounded down.

As a result, the value registered in liquidity (`raw_amount * supply_exchange_price`) can be less than `amount`.

Recommendations:
The `execute_deposit()` function in lending can be modified to use the real liquidity value instead of amount. In the example code below, the `liquidity_amount` is derived from liquidity and used for the calculation of `shares_minted` instead of `amount.` The same can be applied to the `vaults.operate()` implementation.

Rust
```
fn execute_deposit(ctx: Context<Deposit>, amount: u64) -> Result<u64> {
       // Read initial amount directly from account data
       let initial_amount = {
           let account_data = ctx.accounts.lending_supply_position_on_liquidity
              .try_borrow_data()?;
           let position = UserSupplyPosition::try_deserialize(&mut &account_data[8..])?;
           position.get_amount()?
       };

       let mut token_exchange_price: u64 = deposit_to_liquidity(&ctx, amount)?;

       // Read final amount after deposit
       let final_amount = {
           let account_data = ctx.accounts.lending_supply_position_on_liquidity
              .try_borrow_data()?;
           let position = UserSupplyPosition::try_deserialize(&mut &account_data[8..])?;
           position.get_amount()?
       };

       let liquidity_amount = final_amount.saturating_sub(initial_amount);

       token_exchange_price = update_rates(
           &mut ctx.accounts.lending,
           &ctx.accounts.f_token_mint,
           &ctx.accounts.rewards_rate_model,
           token_exchange_price,
       )?;

       // calculate the shares to mint
       // not using previewDeposit here because we just got newTokenExchangePrice
       let shares_minted: u64 = liquidity_amount
           .cast::<u128>()?
           .safe_mul(EXCHANGE_PRICES_PRECISION)?
           .safe_div(token_exchange_price.cast::<u128>()?)?
           .cast::<u64>()?;
```

Customer's response: Fixed in PR.

Fix Review: Fixed for functions `deposit` and `mint`, but not `withdraw` and `redeem`.

M-06 Utilization scaling division done over rounded values		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: user.rs	Status: Fixed	Violated Rules: Last Utilization Invariant

Description:
In liquidity, `utilization` is used in the calculation of supply price from borrow price. Utilization is calculated as follows:

```rust
 utilization = total_borrow
      .safe_mul(FOUR_DECIMALS)?
      .safe_div(total_supply)?
      .cast()?;
```

In this calculation, `total_borrow` and `total_supply` are derived from the total amounts and exchange prices. In this calculation the values are rounded down.
The complete calculation is:

```
utilization =
      (total_borrow_interest_free +
      total_borrow_with_interest * borrow_exchange_price // EXCHANGE_PRICES_PRECISION) //
      (total_supply_interest_free +
      total_supply_with_interest * supply_exchange_price // EXCHANGE_PRICES_PRECISION)
```

Due to rounding, the denominator (`total_supply`) might be rounded down, leading to increase in utilization, in turn, leading to over estimation of the increase in the `supply_exchange_price`

Recommendations:
It is both more efficient and more precise to compute utilization over scaled borrow and supply values:

```
None
utilization =
       (total_borrow_interest_free * EXCHANGE_PRICES_PRECISION +
       total_borrow_with_interest * borrow_exchange_price ) //
       (total_supply_interest_free * EXCHANGE_PRICES_PRECISION +
       total_supply_with_interest * supply_exchange_price)
```

This avoids any rounding in the numerator or denominator.

Customer's response: Fixed in PR.

Fix Review: Fix confirmed.

M-07 Incorrect rounding in liquidate debt

Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: structs.rs	Status: Fixed	Violated Rules: liquidate_integrity_liquidity

Description:
After successful liquidation, the `vault_state.total_borrow` is reduced by total liquidated debt (`total_debt_liq`). The amount the liquidator has to pay is calculated as follows:

```rust
Rust
    fn get_normal_debt_liq(&self, borrow_ex_price: u128) -> Result<u128> {
         Ok(self
             .total_debt_liq
             .safe_mul(borrow_ex_price)?
             .safe_div(EXCHANGE_PRICES_PRECISION)?)
     }
```

In this calculation, the amount is rounded down, meaning that the liquidator pays slightly less than the liquidation value.

Recommendations:
Replace `safe_div` with `safe_div_ceil` for this calculation to enforce rounding up, in favor of the protocol

Customer's response: Fixed in PR.

Fix Review: Fix confirmed.

M-08 Incorrect rounding in calc_revenue		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: token_reserve.rs	Status: Fixed	Violated Rules: liquidity_solvency_collect_revenue

Description:
In collect_revenue, the function calc_revenue is used to compute the revenue of the protocol. The revenue is computed by computing total assets and then subtracting total supply (i.e., assets that are committed). It is better to round up what is being subtracted, but calc_revenue computes total_supply by rounding down. This may cause the protocol to overestimate its revenue, and thereafter drain it more than necessary.

Recommendations:
Replace `safe_div` with `safe_div_ceil` when computing total_supply calculation to enforce rounding up, in favor of the protocol.

Customer's response: Fixed in PR.

Fix Review: Fix confirmed.

M-09 Incorrect rounding direction in unscale debt amount		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: user.rs	Status: Fixed	

Description:
In vault liquidation, the `actual_debt_amount` and `actual_col_amount` are first calculated and then unscaled to adjust for the proper number of decimals. The `unscale_amounts()` function always rounds down, meaning that unscale for tokens that don't have 9 decimals will round down for both `actual_debt_amount` and `actual_col_amount`.
Rounding should always be in favor of the protocol. The `actual_debt_amount` is the amount the liquidator needs to pay and should round up.

Recommendations:
Rewrite the unscaling so it rounds up for `actual_debt_amt`

Customer's response: Fixed in PR.

Fix Review: Fix confirmed.

M-10 Incorrect rounding direction col_per_debt		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: liquidate.rs	Status: Fixed	Violated Rules: [liquidate_solvency]()

Description:
In liquidations, `col_per_debt` is used to calculate the amount of debt that is paid to the liquidator. It is calculated as follows:

```rust
let mut debt_per_col: u128 =
     safe_multiply_divide(exchange_rate, supply_ex_price, borrow_ex_price)?;
...
let raw_col_per_debt: u128 = 10u128
     .pow(RATE_OUTPUT_DECIMALS * 2)
     .safe_div(debt_per_col)?;

let col_per_debt: u128 = raw_col_per_debt
     .safe_mul(FOUR_DECIMALS.safe_add(vault_config.liquidation_penalty.cast()?)?)?
     .safe_div(FOUR_DECIMALS)?;
```

The `debt_per_col` variable computation rounds down. Because the `col_per_debt` divides by this rounded down value, it effectively rounds up.
When doing rounding in favor of the protocol, it should round down.

Recommendations:
Calculate `col_per_debt` directly from `exchange_rate`, `supply_ex_price`, `borrow_ex_price` with rounding direction rounding up

Customer's response: Fixed in [PR]().

Fix Review: Fix confirmed.

M-11 Inconsistent vault updates when actual_debt_amt > debt_amount		
Severity: **Medium**	Impact: **Low**	Likelihood: **Likely**
Files: user.rs structs.rs	Status: Fixed	

Description:
In a liquidation the program calculates the amount of `total_debt_liq` and `total_col_liq` raw amounts that are liquidated. The `get_actual_amounts()` function is called to calculate the net token amounts for these totals that have to be transferred..
Inside `get_actual_amounts()` there is the following codeblock:

```rust
       if actual_debt_amt > debt_amount {
            // Calculate new actual_col_amt via ratio
            actual_col_amt = actual_col_amt
                .safe_mul(debt_amount)?
                .safe_div(actual_debt_amt)?;

            actual_debt_amt = debt_amount;
        }
```

When the `actual_debt_amt` exceeds the `debt_amount` that was supplied by the liquidator, it lowers both `actual_debt_amt` and `actual_col_amt` to match the `debt_amount`.
The total_col_liq and total_debt_liq remain unchanged.

In the liquidate function the vault state is updated as:

```rust
   vault_state.reduce_total_supply(current_data.total_col_liq)?;
   vault_state.reduce_total_borrow(current_data.total_debt_liq)?;
```

This means that in this case the value of repaid debt and collected collateral is not equal to the value of total_debt_liq and total_col_liq, but is slightly less.

Recommendations:
The `actual_debt_amt > debt_amount` branch should be unreachable and it should revert when in case this happens. With proper rounding throughout the liquidation process, the total liquidated value should always be less or equal than the supplied `debt_amount` value.

Customer's response: Fixed in PR.

Fix Review: Fix confirmed.

Low Severity Issues

L-01 Oracle source multiplier configuration is not applied to rate calculation		
Severity: **Low**	Impact: **Low**	Likelihood: **Likely**
Files: structs.rs#L21 helper.rs#L76-L78	Status: Fixed	

Description:

In `get_exchange_rate_for_hop()`, the function retrieves a `rate` and `multiplier` from `read_rate_from_source()`, then calculates the exchange rate as `rate.safe_mul(multiplier)?.saturating_div(source_info.divisor)`. However, the `source_info.multiplier` field from the `Sources` struct is not used in this calculation.

```rust
#[derive(Default, Clone, AnchorSerialize, AnchorDeserialize, InitSpace, Copy)]
pub struct Sources {
    pub source: Pubkey,
    pub invert: bool,
@>> pub multiplier: u128,
    pub divisor: u128,
    pub source_type: SourceType,
}
```

This means the multiplier configuration stored in the Oracle account is ignored, preventing proper rate adjustments when `source_info.multiplier` is set to a non-default value. The divisor is correctly applied, but its corresponding multiplier counterpart is omitted, creating an asymmetric calculation that may produce incorrect exchange rates.

Recommendations:

Apply both the runtime multiplier and the configured source multiplier:

```rust
rate =
rate.safe_mul(multiplier)?.safe_mul(source_info.multiplier)?.saturating_div(source_info.divis
or);
```

Customer's response: Fixed in faf82aa and d9d4be7

Fix Review: Fix confirmed.

L-02 Pyth oracle does not allow future timestamps which can result in DoS when the solana network's clock drifts away even by just a few seconds		
Severity: **Low**	Impact: **Medium**	Likelihood: **Unlikely**
Files: helper.rs#L131-L146	Status: Fixed	

Description:

The code uses different approaches for tracking **stale time** for the different oracles For Chainlink, it's based on slots and **average slot time**, where Redstone and Pyth just check the `timestamp`.

The stale-check function of **Pyth** `get_price_no_older_than` in the SDK allows for **future timestamps**, but our code explicitly disallows that in the `verify_publis_time`:

```
Rust
check!(price.publish_time
                .saturating_add(maximum_age.try_into().unwrap())
@>>             >= clock.unix_timestamp,
            GetPriceError::PriceTooOld
        );
```

For reference if `current_time < publish_time` the `safe_sub` will revert

```
Rust
pub fn verify_publish_time<'info>(publish_time: u64, is_liquidate: Option<bool>) ->
Result<()> {
    let current_time = Clock::get()?.unix_timestamp as u64;

@>>  // @dev reverts if current_time < publish_time
```

```
    if is_liquidate.is_some() && is_liquidate.unwrap() {
        if current_time.safe_sub(publish_time)? > MAX_AGE_LIQUIDATE {
            return err!(ErrorCodes::PriceTooOld);
        }
    } else {
@>>     if current_time.safe_sub(publish_time)? > MAX_AGE_OPERATE {
            return err!(ErrorCodes::PriceTooOld);
        }
    }

    Ok(())
}
```

Even though there are constraints of how likely it is for this to happen, on **Solana** it's possible that the **Clock** can drift away from the real-world clock, when slots take more than the target of **400ms**. In such a case, if the `timestamp` coming from the price feed is very recent, it could end up in the future relative to Solana's clock and result in a DoS of oracle-dependent functions.

A crucial difference is how the `timestamp` provided by each oracle is retrieved. For Pyth is **off-chain generated**, while for Redstone, it's **on-chain generated**. Thus for **Redstone** this issue can't occur. Chainlink also can't end up in this case, as we don't use timestamp for its stale check.

There are historical cases of Solana's clock drifting backwards:
https://www.theblock.co/post/149112/solanas-blockchain-clock-loses-track-of-time-now-running-30-minutes-behind

https://medium.com/@observer1/solana-drift-53d36d1aca1

However since then changes were applied which significantly reduce the likelihood of sustained drift, refer to the Solana docs:
https://docs.solanalabs.com/implemented-proposals/validator-timestamp-oracle
https://docs.solanalabs.com/implemented-proposals/bank-timestamp-correction

Currently we use stake-weighted timestamp, however we still advance the time based on the **400ms** target, refer to:
https://github.com/anza-xyz/agave/blob/9f4b3ae012c1ee17f0957537eeb8c162b5bc7d5c/runtime/src/stake_weighted_timestamp.rs#L44-L45

So in more extreme cases where there is a voting delay from big-stake validators combined with longer slot times it is theoretically possible to still result in a sustained drift, where even a few seconds are enough to cause this issue.

The attack path (low) is the following:

1. Last-second price from Pyth is submitted on-chain.
2. The latest price's timestamp is always in the future because Solana's clock drifts back.
3. Liquidations or other oracle-dependent operations are **DoS-ed**, which are usually time-sensitive.

Pyth oracle has a pull-based mechanism as it allows basically anyone to submit the latest price at any time. (The issue could also happen naturally or by an adversary constantly submitting updates)

Crypto feeds https://insights.pyth.network/price-feeds?assetClass=Crypto from Pyth are insta-updated on the pyth network (and their relative `publish_time` also, this could be verified with a request to Hermes https://docs.pyth.network/price-feeds/core/fetch-price-updates), so this won't require too big of a drift.

Recommendations:

Allow room for future timestamps, within acceptable limits. Or adopt the approach of the `get_price_no_older_than` function.

Customer's response: Fixed

Fix Review:
The pyth oracle now uses the `get_price_no_older_than` function which solves the issue

L-03 Mint and metadata accounts are not closed, permanently locking rent funds		
Severity: **Low**	Impact: **Low**	Likelihood: **Likely**
Files: user.rs#L47-L69	Status: Acknowledged	

Description:

When a position is closed via `close_position()` , it performs a CPI call to Metaplex's `BurnV1` to burn the single NFT token and close the token account. However, this operation does not close the underlying mint account or the metadata account. After the burn completes, the mint has a supply of 0 and could be closed to recover rent, but the code does not perform this cleanup.

Net SOL lost per position lifecycle:

- Mint account rent: `rent.minimum_balance(82)` = 0.0014616 SOL
- Metadata account rent: `rent.minimum_balance(607)` = 0.0051156 SOL
- Total permanently locked: 0.0065772 SOL (~$1 at SOL = $150)

The Metaplex creation fee of ~0.01 SOL is already a sunk cost regardless of whether accounts are closed. By not closing these accounts, users lose the rent deposits that should be recoverable. Over many position lifecycles, this represents significant accumulated losses for users.

Recommendations:

Adopt SPL Token-2022 with the `MintCloseAuthority` extension to enable closing mints after mint authority removal:

1. **For new positions:** Initialize the mint using Token-2022 with `MintCloseAuthority` extension set to `vault_admin`. This allows closing the mint in `close_position()` even after the mint authority is removed.
2. **In `close_position()`:** Add logic to close both the mint and metadata accounts:
   - Check if the mint owner is Token-2022 program and have `MintCloseAuthority` extension → close both mint and metadata

    - Check if the mint owner is Token-2022 program and doesn't `MintCloseAuthority` extension → only close metadata
    - If the mint owner is standard Token Program → only close metadata (mint cannot be closed in standard Token Program)
3. **Migration consideration:** Implement dual support to handle existing positions created with the standard Token Program while transitioning new positions to Token-2022.

This would recover ~0.0066 SOL to users for each position lifecycle, which accumulates significantly over many position operations.

Customer's response: Acknowledged

Informational Severity Issues

I-01 Oracle exponent exceeding rate decimals causes underflow

Description:

When normalizing oracle prices to the protocol's 15-decimal standard in **read_rate_from_source()** function, the code computes delta = `10u128.pow(RATE_OUTPUT_DECIMALS.safe_sub(multiplier)?)` where the **multiplier** is the oracle's exponent.

For feeds where the decimals are above 15, this subtraction (15-18) underflows and reverts, preventing these oracles from being used:

```rust
SourceType::Chainlink => {
   let chainlink_price = read_chainlink_source(&source, is_liquidate)?;
   let multiplier: u32 = chainlink_price.exponent.unwrap().cast()?;
   let delta = 10u128.pow(RATE_OUTPUT_DECIMALS.safe_sub(multiplier)?);
            (chainlink_price.price, delta)
        }
```

This issue exists for Chainlink, Redstone and Pyth sources, which results in blocking future legitimate feeds from integration. Currently only custom feeds are expected to exceed 15 decimals and such setup has to be done by an admin.

Recommendations:
Handle cases where the exponent meets/exceeds **`RATE_OUTPUT_DECIMALS`**.

Remove:

```rust
let delta = 10u128.pow(RATE_OUTPUT_DECIMALS.safe_sub(multiplier)?); - (chainlink_price.price,
delta)
```

And replace with:

```rust
if multiplier >= RATE_OUTPUT_DECIMALS {

    let rate =
chainlink_price.price.safe_div(10u128.pow(multiplier.safe_sub(RATE_OUTPUT_DECIMALS)?))?;

    (rate, 1)

 } else {

    let delta = 10u128.pow(RATE_OUTPUT_DECIMALS.safe_sub(multiplier)?);

    (chainlink_price.price, delta)

 }
```

Apply similar logic to Pyth and Redstone cases for future-proofing.

Customer's response: Fixed in faf82aa and d9d4be7

Fix Review: Fix confirmed

certora

Formal Verification

Verification Methodology

We performed verification of the **Fluid/JupiterLend** protocol protocol using the Certora verification tool which is based on Satisfiability Modulo Theories (SMT).
In short, the Certora verification tool works by compiling formal specifications written in the Certora Verification Language (CVLR) and the implementation source code written in Rust.
More information about Certora's tooling can be found in the Certora Technology Whitepaper.

If a property is verified with this methodology it means the specification in CVLR holds for all possible inputs. However specifications must introduce assumptions to rule out situations which are impossible in realistic scenarios (e.g. to specify the valid range for an input parameter). Additionally, SMT-based verification is notoriously computationally difficult. As a result, we introduce overapproximations (replacing real computations with broader ranges of values) and underapproximations (replacing real computations with fewer values) to make verification feasible.

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 a smart contract. We use constructors/initialization functionality as a base case, and consider all other (relevant) externally callable functions that can change the storage 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 this computation any resulting state satisfies the invariant. Each such case results in one rule.

In the following, we provide the verification plan for each contract in their individual section. We provide the specification, the assumptions and finally, the verification results.

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

- Configuration:
  - Solana Platform Tools - v1.43.
  - Loops are unrolled at most once.
- In all formally verified protocols, the SafeMath library has been substituted to perform the same arithmetic as before, but via Certora Prover NativeInts Library instead of u64/u128/u256. Certora's NativeInts Library is already formally verified, and translates directly to the underlying SMT reasoning. This reduces the processing time for verification tasks.

Liquidity Contract

Verification Plan

The formal verification of the Liquidity contract focuses around proving solvency of the token reserve. We state the solvency as an invariant of the Liquidity contract. We begin by defining the following terms:

`TokenSupply_wi:` `token_reserve.total_supply_with_interest`

`TokenSupply_if:` `token_reserve.total_supply_interest_free`

`TokenBorrow_wi:` `token_reserve.total_borrow_with_interest`

`TokenBorrow_if:` `token_reserve.total_borrow_interest_free`

`SupplyPrice:` `token_reserve.supply_exchange_price`

`BorrowPrice:` `token_reserve.borrow_exchange_price`

`TokenTotalClaim:` `token_reserve.total_claim_amount`

```
TokenTotalSupply: TokenSupply_if * EXCHANGE_PRICE_PRECISION

                    + TokenSupply_wi * SupplyPrice
```

```
TokenTotalBorrow: TokenBorrow_if * EXCHANGE_PRICE_PRECISION

                    + TokenBorrow_wi * BorrowPrice
```

`VaultBalance:` `balanceOf(TokenReserve.vault)`

With the above terms defined, we can define liquidity solvency invariant as follows:

```
TokenTotalSupply + TokenTotalClaim <= VaultBalance + TokenTotalBorrow
```

The LHS represents the amount (in actual units) that the token reserve owes to its suppliers. The RHS represents the amount that the token reserve currently has plus the amount expected from the borrowers. The solvency of the token reserve simply states that the token reserve must expect to receive more than it owes to its suppliers.

We prove the solvency invariant by establishing that all functions of the Liquidity contract preserve it. Since admin functions do not change the above terms, we focus on `operate` and `update_exchange_price`. Particularly for `operate`, we check for deposit, withdraw, borrow and payback code paths individually. This forms the first property:

(P-1) Solvency Invariant is preserved by `operate`, `update_exchange_price`, `claim` and `collect_revenue`.

In order to establish solvency invariant, the following property must also be established:

(P-2) The token reserve stores `last_utilization` correctly.

This is also proven as an invariant over `operate` and `update_exchange_price`. We call this the "Last Utilization Invariant". Due to the issue reported in M-06, this invariant did not hold initially, but holds after fixes.

In addition to solvency invariants, following integrity properties are crucial to correct functioning of the liquidity contract:

(P-3a) `TokenSupply_wi` is the sum of all `user_supply_positions.amount` for all user supply positions with interest. Similar is true for `TokenSupply_if`.

(P-3b) `TokenBorrow_wi` is the sum of all `user_borrow_positions.amount` for all user borrow positions with interest. Similar is true for `TokenBorrow_if`.

(P-3c) `TokenTotalClaim` amount must be the sum of all `user_claim.amount`.

We prove property (P3a-c) over `operate`.

<u>**General Assumptions**</u>

- When proving properties (P1-3) over `operate`, we assume that the prices are already updated. This is justified because the correctness of price update logic is checked when proving solvency invariant for `update_exchange_price`.
- When proving the solvency invariant for `update_exchange_price`, we assume that `total_supply_interest_free` and `total_borrow_interest_free` are zero. This is justified because only the Flashloan contract is set without interest. Note that, Flashloan

contract updates prices before it is active, and does not allow price updates once it is active. Additionally, the flashloan must be paid off at the end of the transaction. Hence, the interest free quantities are set to zero at the end.

- The parameters of `operate` are `i64` instead of `i128`.
- The RateModel function `calc_borrow_rate_from_utilization` is mocked to return nondeterministic values.

Program Properties

(P-01) Liquidity Solvency Invariant

Status: Verified after fix

Rule Name	Status	Description	Links
liquidity_solvency_operate_deposit	Verified	*Case: operate_deposit*	*Rule report* *Rule report after fix*
liquidity_solvency_operate_withdraw	Verified	*Case: operate_withdraw*	
liquidity_solvency_operate_borrow	Verified	*Case: operate_borrow*	
liquidity_solvency_operate_payback	Verified	*Case: operate_payback*	
liquidity_price_update_only_with_interest **update_price_fee_period_lemma**	Verified after fix	*Case: update_exchange_price. Note that the rule uses a lemma that fails due to M-03.*	
liquidity_solvency_claim	Verified	*Case: claim*	
liquidity_solvency_collect_revenue	Verified after fix	*Case: collect_revenue. Violation due to issue M-08.*	

(P-02) Last Utilization Invariant

Status: Verified after fix

Rule Name	Status	Description	Links
liquidity_operate_deposit_utilization	Verified after fix	*Case: operate_deposit. Violation due to M-06.*	*Rule report* *Rule report after fix*
liquidity_operate_withdraw_utilization	Verified after fix	*Case: operate_withdraw. Violation due to M-06.*	
liquidity_operate_borrow_utilization	Verified after fix	*Case: operate_borrow. Violation due to M-06.*	
liquidity_operate_payback_utilization	Verified after fix	*Case: operate_payback. Violation due to M-06.*	
liquidity_price_update_utilization	Verified after fix	*Case: update_exchange_price. Violation due to M-06.*	

certora

(P-03) Operate Integrity Properties

Status: Verified	

Rule Name	Status	Description	Links
operate_integrity _deposit_or_withdraw	Verified	*Function operate satisfies property P-3a in case of deposit or withdraw.*	[Rule report]()
operate_integrity _borrow_or_payback	Verified	*Function operate satisfies property P-3b in case of borrow or payback.*	
operate_borrow _claim_integrity	Verified	*Function operate satisfies property P-3c in case of borrow.*	
operate_withdraw _claim_integrity	Verified	*Function operate satisfies property P-3c in case of withdraw.*	

Lending Contract

Verification Plan

The formal verification of the Lending contract focuses on proving its solvency. This involves comparing the obligations that Lending owes to its users (depositors) versus the assets it owns at the liquidity layer. We begin by defining the following terms:

`FTokenSupply`: `lending.f_token_mint.supply`

`FTokenPrice`: `lending.token_exchange_price`

`SupplyPrice` : `token_reserve.supply_exchange_price` (`token_reserve` for lending's underlying asset)

`LendingPositionAmount`: `lending_user_supply_position.amount`

`LendingAssets`: `FTokenSupply * FTokenPrice`

`LiquidityAssets`: `LendingPositionAmount * SupplyPrice`

Note that, `LendingAssets` represents the assets that Lending owes to its users. On the other hand, `LiquidityAssets` represents the assets that belong to the Lending contract at the liquidity layer. Solvency can be stated as:

`LendingAssets <= LiquidityAssets`.

However, due to the issue discussed in M-05, it becomes clear that solvency in above terms does not hold. In such a situation, it is useful to analyze the difference in assets, i.e.

`LendingAssets - LiquidityAssets`

We would like to determine the upper bound of how much the delta between the two assets can grow with one call to any Lending function. If before a function call,

`LendingAssets - LiquidityAssets <= k`

holds for some k, then after the function call

`LendingAssets' - LiquidityAssets' <= k + Bound` must hold. Here, LendingAssets' and LiquidityAssets' represent the value of those terms after the function call. The Certora team determined this bound and mathematically proved it with the Certora prover. The bound is as follows:

Bound = SupplyPrice + FTokenSupply * (FTokenPrice' - FTokenPrice)

Intuitively, the Lending contract loses value equal to the sum of two components: (i) the price of one share of liquidity due to the issue in M-05, (ii) the additional value it offers to existing FToken holders due to increase in FTokenPrice. Note that the second component only comes into play if rewards are set for the Lending program. It was manually reviewed that the second component matches the rewards set by the LendingRewardsRateModel. Finally, the property we prove is the following:

(P-4) (Bound on assets delta) LendingAssets' - LiquidityAssets' <=

LendingAssets - LiquidityAssets + Bound.

We prove this for the functions deposit, mint, redeem and withdraw. We ignore the equivalent functions with slippage protection as they call the same internal functions.

Note after fix: the Bound after fix is determined to be FTokenSupply * (FTokenPrice' - FTokenPrice), since issue M-05 is resolved.

General Assumptions

- The liquidity prices stored at the Lending program are already updated. Similarly, the token_reserve prices are already updated before the function call. This is justified because these price updates are analyzed separately.
- The mint, burn and transfer functions are replaced with already formally verified CVLR versions of these functions.
- The call to liquidity operate from lending is mocked to modify only the user_supply_position, as the property (P-4) only cares about the position of Lending program at liquidity.
- calculate_new_token_exchange_price has been over-approximated to return a nondeterministic price between 1*EXCHANGE_PRICE_PRECISION and 2*EXCHANGE_PRICE_PRECISION.

certora

<u>Program Properties</u>

(P-04) Bound on assets delta	
Status: Verified	

Rule Name	Status	Description	Links
lending_solvency_deposit	Verified	*Case: deposit.*	[*Rule report*]
lending_solvency_mint	Verified	*Case: mint*	
lending_solvency_redeem	Verified	*Case: redeem*	
lending_solvency_withdraw	Verified	*Case: withdraw*	

Vaults Contract

Verification Plan

To define solvency of the Vaults contract, we define the following terms:

1. `TotalSupply`: `vault_state.total_supply`
2. `TotalBorrow`: `vault_state.total_borrow`
3. `LiquiditySupply`: `vault_user_supply_position.amount`
4. `LiquidityBorrow`: `vault_user_borrow_position.amount`
5. `CurrentUserSupply`: `supply of an up-to-date position`
6. `SumCurrentUserSupply`: `sum of supply of an up-to-date position for all positions`
7. `CurrentUserDebt`: `debt of an up-to-date position`
8. `SumCurrentUserDebt`: `sum of debt of an up-to-date position for all positions`
9. `BranchDebt`: `amount of debt stored in a branch`
10. `TickDebt`: `amount of debt stored in a tick`
11. `SumBranchDebt`: `sum of amount of debt stored in a branch over all branches`
12. `SumTickDebt`: `sum of amount of debt stored in a tick over all ticks`
13. `VaultSupplyPrice`: `vault_state.vault_supply_ex_price`
14. `VaultBorrowPrice`: `vault_state.vault_borrow_ex_price`
15. `LiquiditySupplyPrice`: `supply_token_reserve.supply_exchange_price`
16. `LiquidityBorrowPrice`: `borrow_token_reserve.borrow_exchange_price`
17. `OracleSupplyPrice`: `oracle supply price`
18. `OracleBorrowPrice`: `oracle borrow price`

The following properties define solvency of the vaults contract:

(P-A) Relating vault aggregate with liquidity:

```
- VaultSupplyPrice * TotalSupply <= LiquiditySupplyPrice *
  LiquiditySupply
- VaultBorrowPrice * TotalBorrow >= LiquidityBorrowPrice *
  LiquidityBorrow
```

(P-B) Relating vault aggregate with user positions:

```
- TotalSupply >= SumCurrentUserSupply
```

- `TotalBorrow <= SumCurrentUserDebt`

(P-C) Relating vault aggregate with branches and ticks:

```
TotalBorrow == SumBranchDebt + SumTickDebt
```

(P-D) Relating branches with user positions:

```
BranchDebt == sum_{all positions that point to that branch}
CurrentUserDebt

TickDebt == sum_{all positions that point to that tick}
CurrentUserDebt
```

(P-E) Vault Solvency:

```
VaultSupplyPrice * TotalSupply * OracleSupplyPrice >=

     VaultBorrowPrice * TotalBorrow * OracleBorrowPrice
```

We prove properties (P-A) to (P-E) for functions `operate` and `liquidate`.

General Assumptions

- The call to liquidity `operate` from vaults is mocked to modify only the `user_supply_position` and `user_borrow_position`, as the property (P-A) only cares about the position of vaults program at liquidity.
- Both borrow and supply tokens are assumed to be 9 decimals. Thus, `scale` and `unscale` functions do not change the input.
- For `operate`, we assume the user position is already up-to-date. The code path which updates the position was reviewed manually.
- For `liquidate`, the FV is set up such that there is exactly one tick above liquidation threshold. We also ignore code paths that absorb the debt. All code paths not covered under FV were reviewed manually.

- We use a summary of the `get_new_position_info` function. This summary is verified by rule `verify_get_new_position_info_lemma`.
- Function `check_if_position_safe` is assumed to be nondeterministic, except for rule `operate_position_safe_tick`.
- For all rules involving a user position except `operate_position_safe_tick`, the initial tick is assumed to be 1 and final tick is assumed to be between 0 and 2. Thus, a position's tick can move in either direction, allowing us to catch bugs in either direction.
- For rule `operate_position_safe_tick`, collateral factor ratio is fixed at 2 with `oracle_exchange_price` (of supply in terms of borrow) at 2 and vault supply and borrow prices at 1. The corresponding collateral factor tick is 462.
- Liquidity prices are assumed to be 1 (in 1e12), vault prices are assumed to be nondeterministically bigger than 1 (in 1e12). This allows us to simplify computation in liquidity, while allowing us to catch rounding errors in vaults.
- Following functions are assumed to return nondeterministic values: `mint_position_token_and_remove_authority`, `check_if_withdrawal_safe_for_withdrawal_gap`, `check_if_ratio_safe_for_deposit_or_payback`, and `get_debt_liquidated`.

Program Properties

(P-05) Correctness of Vaults operate.

Status: Verified

Rule Name	Status	Description	Links
operate_integrity_liquidity	Verified	*Check (P-A) for operate.*	Rule Report
operate_integrity_position	Verified	*Check (P-B) for operate*	Rule Report Rule Report
operate_integrity_tick	Verified	*Check (P-C) and (P-D) for operate*	
operate_vault_totals **operate_position_safe_tick** **safe_tick_implies_position _solvency** **vault_solvency_lemma**	Verified	*Check (P-E) for operate. The proof is decomposed into following steps:* *(i) operate_vault_totals: ensures vault total borrows and supplies are updated accordingly.* *(ii) operate_position_safe_tick: operate ensures the position's tick is in safe zone (below collateral factor).* *(iii) safe_tick_implies_position_solvency: if a position's tick is in safe zone, then it is solvent.* *(iv) vault_solvency_lemma: if each position is solvent, then the vault is solvent.*	
verify_get_new_position _info_lemma	Verified	*Verify the summary of get_new_position_info.*	

(P-06) Correctness of Vaults liquidate.

Status: Verified after fix

Rule Name	Status	Description	Links
liquidate_integrity_liquidity	Verified after fix	*Check (P-A) for liquidate. Violation due to M-07.*	*Rule Report* *Rule Report after fix*
liquidate_respects_slippage	Verified	*Check that liquidation respects slippage*	*Rule Report*
liquidate_integrity_tick_branch	Verified	*Check (P-C) and (P-D) for liquidate.*	
liquidate_solvency	Verified after fix	*Check (P-E) for liquidate. Violation due to M-10.*	*Rule Report* *Rule Report after fix*

Big Number Library

Overview: **big_number ([bn.rs](bn.rs))**

These are the numbers used to represent `debt_factor`, and `connection_factor`, for `Tick` and `Branch.` The numbers are defined in `library/src/bn.rs`, however, they are specific to `Vaults`.

A `big_num` is a `u64` that represents a tuple `(m, e)`, where matissa `m` is 35 bits and exponent `e` is 15 bits. A big_num `(m, e)` represents the value `m * 2^(e - 16384).` There is a requirement that the 35th bit of `m` is 1. Note that `16384 = 2^14` so effectively the high bit of `e` indicates a sign, with `1` being positive and `0` negative. We write `val(num)` for the value of a big_num `num`

Operations are:

- `mul_div_normal(normal, big_num1, big_num2) -> normal:` Computes `normal * val(big_num1) / val(big_num2)`
  - this is used to compute debt of a position after liquidation
  - rounds down
  - `big_num1` is expected to be branch debt factor
  - `big_num2` is expected to be the connection factor
  - these numbers have expected ranges, this is important
- `mul_div_big_number(big_num, normal) -> big_num:` computes `val(big_num) * normal / 2^64`
  - used to adjust `debt_factor` during liquidation
  - `big_num` is the initial `debt_factor`
  - `normal` is the result of `new_debt / old_debt` scaled by `2^64`
  - the division by `2^64` is to remove the scale mentioned above
  - `debt_factor` is scaled by `X == (2^35 - 1) * 2^14`
  - `debt_factor` is `X` initially, and is then reduced using `mul_div_big_number` on every liquidation step
- `mul_big_number(big_num1, big_num2) -> big_num:` multiplies two big nums. Used to combine `connection_factor`

    - connection_factor of the Tick, called branch_factor in Tick, is scaled by X
    - connection_factor of a merged branch is not scaled
    - sequence of mul_big_number always starts with a number scaled by X and the final result is scaled by X
- div_big_number(big_num1, big_num2) -> big_num: divides two big nums. This is used to compute connection_factor.
    - big_num1 is the branch_factor of the branch being merged into.
    - big_num2 is a branch_factor of the current branch. It is scaled by X.
    - The result is expected to be unscaled.

All operations round down. For that reason, connection_factor can be either smaller or larger than expected. If connection_factor is smaller, protocol loses funds.

In short,

- big_num represents numbers with 35 significant bits;
- debt_factor is scaled by X (so X is 1 in debt_factor);
- exactly one connection_factor is expected to be scaled by X, so that when debt_factor is divided by an accumulated connection_factor, X cancels out and the result is a fraction
- during liquidation, debt reduction ratio is represented with 64 bit of precision

Some tests in library fail:

```
Shell
$ cd crates/library ; cargo test -- --nocapture
failures:
    math::bn::tests::test_precision_consistency
    math::tick::tests::test_bounds
```

Failure of test_precision_consistency is due to handling of zero.

Error Model

Big numbers are specialized floating point numbers. The error model is similar to the floating point number model, but without infinity, NaN, and sub-normals.

The unit roundoff `u = 2^{-35} = 2.91e-11`.

A rounding error for an operation `f(a,b)` is relative to the magnitude of the result, the roundoff error, and is always flooring.

`f'(a, b) = f(a, b)(1 - δ)`, where `0 <= δ <= u`.

The roundoff of `k` operations is linear in `k`:

`f'(a,b)^k = f(a,b)^k(1 - kδ)`

For example,

```Rust
x = bn(0x400000001, 16384)
y = bn(0x7ffffffff, 16384)
z = x*y = bn(0x200000000000000000, 16384) = 590295810358705651712
z_math = 0x400000001 * 0x7ffffffff = 590295810375885520895
z_math - z = 17179869183 < u * z
```

Rounding

All bn operations round down. We analyze the effect of the rounding for each individual operation based on its use in the vaults code

1. `mul_div_normal` computes the raw debt of the position after liquidation.

```Rust
                position_raw_debt = mul_div_normal(
                    position_raw_debt.cast()?,
                    branch_min_debt_factor.cast()?,
                    current_connection_factor.cast()?,
                )?
                .cast()?;
```

rounding down means that position_raw_debt is lower than it was expected. Therefore position_raw_col is proportionally lower as well. This is a loss to the user, but a win to the protocol. Furthermore, the check for whether a position was reduced to less than 1% might be skipped. In this case, the user avoid complete liquidation. The rounding of position_raw_debt itself is by 1 because position_raw_debt is a normal number.

2. mul_div_big_number is used to compute debt_factor after liquidation

```rust
self.debt_factor = mul_div_big_number(self.debt_factor, debt_factor.cast()?)?;
```

Rounding down reduces debt factor.

3. mul_big_number is used to combine connection factors

```rust
                current_connection_factor = mul_big_number(
                    current_connection_factor.cast()?,
                    current_branch.debt_factor.cast()?,
                )?
                .cast()?;
```

Rounding down reduces connection_factor. Since connection factor is used to divide debt factor (see mul_div_normal), smaller connection factor increases position_raw_debt. This is away from the protocol. That is, it is a loss to the protocol. connection_factor is a big number, so the absolute error can be large. However, relative error is small, so it should not affect division significantly.

4. div_big_number is used to convert a branch factor into a connection factor by dividing two branch factors:

```Rust
                    current_branch.merge_with_base_branch(div_big_number(
                        new_branch_debt_factor,
                        branch.debt_factor,
                    )?)?;
```

Rounding down makes the connection factor smaller. Connection factors are expected to be small in magnitude, so the error is small both in absolute and relative terms. Connection factor is used to divide branch factor (see `mul_div_normal`). This is similar to `mul_big_number`.

Recommendations

1. Use `safe_math` instead of relying on compiler flags for proper handling of overflow. For example, in `mul_div_normal`,

```Rust
// Calculate result and check for overflow
let result = numerator / denominator;
if result > u64::MAX as u128 {
    return Err(error!(ErrorCodes::LibraryBnError));
}
```

should be changed to

```Rust
let result = numerator.safe_div(denominator)?;
result.try_into().map_err(|_| error!(ErrorCodes::LibraryBnError))
```

This provides better error messages in case of failure, independent of compilation flags and compiler behavior, and is easier to audit.

2. Explicitly check that if a `big_number` is not 0, then it satisfies the requirement that high bit of coefficient is always one.

Rust

```rust
    if coefficient < COEFFICIENT_MIN {
      return Err(error!(ErrorCodes::LibraryBnError));
   }
```

This will prevent any future mistakes when an incorrect big number flows into the computation when it is not expected. The check has to be added only after the special handling of 0.

3. In `mul_div_normal`
   - fix handling of multiplication and division by zero (see additional tests)
   - fix handling of division by a large number (see additional test)
   - remove incorrect "Check for division by zero"
4. in `mul_div_big_number`
   - fix handling of multiplication by a normal 0 (see additional test)
   - use `checked_sub` instead of `saturating_sub`:

Rust

```rust
// Calculate difference in bits to make the result_numerator 35 bits again
diff = diff.saturating_sub(COEFFICIENT_SIZE_DEBT_FACTOR);
```

   - use `checked_add` instead of `saturating_add`:

Rust

```rust
// Calculate new exponent
let result_exponent = exponent.saturating_add(diff as u64);
```

5. in `mul_big_number`
   - fix handling of multiplication by 0 (see additional tests)
6. in `div_big_number`
   - fix handling division by 0 (see additional tests)
7. It is advisable to wrap big numbers by a proper Rust type, different from `u64`. For example, by creating `struct BigNumber(u64)` . This will make the compiler catch any potential error when `u64` is used incorrectly when big number is expected.

Some extra tests:

```rust
#[test]
    fn test_mul_div_normal_zero1() {
        let normal = 42;
       let big_number1 = 0;
       let big_number2 = create_big_number(COEFFICIENT_MIN, 16384);

       let result = mul_div_normal(normal, big_number1, big_number2);
       assert_eq!(result, Ok(0));

   }

    #[test]
    fn test_mul_div_normal_zero2() {
        let normal = 42;
       let big_number1 = 0;
       let big_number2 = create_big_number(COEFFICIENT_MIN, 16384);

       let result = mul_div_normal(normal, big_number2, big_number1);
       assert!(result.is_err());
   }

    #[test]
    fn test_mul_div_normal_zero3() {
        let normal = 42;
       let big_number1 = 0;

       let result = mul_div_normal(normal, big_number1, big_number1);
       assert!(result.is_err());
   }

    #[test]
    fn test_mul_div_normal_zero4() {
        let normal = 0;
       let big_number1 = 0;

       let result = mul_div_normal(normal, big_number1, big_number1);
       assert!(result.is_err());
   }

    #[test]
    fn test_mul_div_normal_zero5() {
        let normal = 0;
       let big_number1 = create_big_number(COEFFICIENT_MIN, 16384);

       let result = mul_div_normal(normal, big_number1, big_number1);
       dbg!(normal, big_number1, &result);
       assert_eq!(result, Ok(0));
   }

    #[test]
    fn test_mul_div_normal_large() {
        let normal = u64::MAX;
       let big_number1 = create_big_number(COEFFICIENT_MAX, 16384);
       let big_number2 = create_big_number(COEFFICIENT_MIN + 1, 16384 + 94);

       let result = mul_div_normal(normal, big_number1, big_number2);
       dbg!(normal, big_number1, big_number2, &result);
       assert_eq!(result, Ok(0));
   }

    #[test]
    fn test_mul_div_big_number_zero1() {
       let big_number1 = 0;
       let normal = 42;

       let result = mul_div_big_number(big_number1, normal);
       assert_eq!(result, Ok(0));
   }

    #[test]
    fn test_mul_div_big_number_zero2() {
       let big_number1 = create_big_number(COEFFICIENT_MIN, 16384);
       let normal = 0;

       let result = mul_div_big_number(big_number1, normal);
       assert_eq!(result, Ok(0));
   }

  #[test]
  fn test_mul_big_number_zero1() {
       let big_number1 = create_big_number(COEFFICIENT_MIN, 16384);
       let big_number2 = 0;
       let result = mul_big_number(big_number1, big_number2);
       assert_eq!(result, Ok(0));
    }

    #[test]
    fn test_mul_big_number_zero2() {
       let big_number1 = 0;
       let big_number2 = create_big_number(COEFFICIENT_MIN, 16384);

       let result = mul_big_number(big_number1, big_number2);
       assert_eq!(result, Ok(0));
    }

  #[test]
    fn test_mul_big_number_zero3() {
       let big_number1 = 0;
       let big_number2 = 0;

       let result = mul_big_number(big_number1, big_number2);
       assert_eq!(result, Ok(0));
    }

  #[test]
    fn test_div_big_number_zero1() {
       let big_number1 = 0;
       let big_number2 = 0;

       let result = div_big_number(big_number1, big_number2);
       assert!(result.is_err());
    }

  #[test]
    fn test_div_big_number_zero2() {
       let big_number1 = create_big_number(COEFFICIENT_MIN, 16384);
       let big_number2 = 0;

       let result = div_big_number(big_number1, big_number2);
       assert!(result.is_err());
    }

  #[test]
```

```
    fn test_div_big_number_zero3() {
        let big_number1 = 0;
        let big_number2 = create_big_number(COEFFICIENT_MIN, 16384);

        let result = div_big_number(big_number1, big_number2);
        assert_eq!(result, Ok(0));
    }
}
```

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.