certora

Security Assessment

exponent

Exponent – Core Program

June 2025

Prepared for Exponent Finance

certora exponent

Table of Contents

certora exponent

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform
exponent-core	https://github.com/exponent-finance/exponent-core	1153aae	Solana

Project Overview

This document describes the manual code review of the **Exponent Core** program

The work was a 5.5 weeks effort undertaken from **28/04/2025** to **05/06/2025**

The following contract list is included in our scope:

- src/allocator.rs
- src/error.rs
- src/instructions/market_two/admin/add_farm.rs
- src/instructions/market_two/admin/add_market_emission.rs
- src/instructions/market_two/admin/market_two_init.rs
- src/instructions/market_two/admin/mod.rs
- src/instructions/market_two/admin/modify_farm.rs
- src/instructions/market_two/admin/modify_market_setting.rs
- src/instructions/market_two/buy_yt.rs
- src/instructions/market_two/claim_farm_emissions.rs
- src/instructions/market_two/deposit_liquidity.rs
- src/instructions/market_two/deposit_lp.rs
- src/instructions/market_two/init_lp_position.rs
- src/instructions/market_two/market_collect_emission.rs
- src/instructions/market_two/mod.rs

- src/instructions/market_two/sell_yt.rs
- src/instructions/market_two/trade_pt.rs
- src/instructions/market_two/withdraw_liquidity.rs
- src/instructions/market_two/withdraw_lp.rs
- src/instructions/mod.rs
- src/instructions/self_cpi.rs
- src/instructions/util.rs
- src/instructions/vault/admin/add_emission.rs
- src/instructions/vault/admin/initialize_vault.rs
- src/instructions/vault/admin/mod.rs
- src/instructions/vault/admin/modify_vault_setting.rs
- src/instructions/vault/admin/treasury/collect_treasury_emission.rs
- src/instructions/vault/admin/treasury/collect_treasury_interest.rs
- src/instructions/vault/admin/treasury/mod.rs
- src/instructions/vault/collect_emission.rs
- src/instructions/vault/collect_interest.rs
- src/instructions/vault/common.rs
- src/instructions/vault/deposit_yt.rs
- src/instructions/vault/initialize_yield_position.rs
- src/instructions/vault/merge.rs
- src/instructions/vault/mod.rs
- src/instructions/vault/stage_yield.rs
- src/instructions/vault/strip.rs
- src/instructions/vault/tests.rs
- src/instructions/vault/withdraw_yt.rs
- src/instructions/wrappers/buy_pt.rs
- src/instructions/wrappers/mod.rs
- src/instructions/wrappers/sell_pt.rs
- src/instructions/wrappers/wrapper_buy_yt.rs
- src/instructions/wrappers/wrapper_collect_interest.rs
- src/instructions/wrappers/wrapper_merge.rs
- src/instructions/wrappers/wrapper_provide_liquidity_base.rs
- src/instructions/wrappers/wrapper_provide_liquidity_classic.rs
- src/instructions/wrappers/wrapper_provide_liquidity.rs
- src/instructions/wrappers/wrapper_sell_yt.rs

- src/instructions/wrappers/wrapper_strip.rs
- src/instructions/wrappers/wrapper_withdraw_liquidity_classic.rs
- src/instructions/wrappers/wrapper_withdraw_liquidity.rs
- src/lib.rs
- src/seeds.rs
- src/state/cpi_common.rs
- src/state/lp_position.rs
- src/state/market_two.rs
- src/state/mod.rs
- src/state/personal_yield_tracker.rs
- src/state/vault.rs
- src/state/yield_token_position.rs
- src/utils/math.rs
- src/utils/mod.rs
- src/utils/pda.rs
- src/utils/sy_cpi.rs

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

Protocol Overview

Exponent's Core program is a decentralized finance (DeFi) protocol designed to facilitate yield-based swaps on the Solana blockchain. It creates a market for yield expectations by splitting yield-bearing positions into two components, allowing users to trade fixed and floating yield exposures.

Core Concept and Architecture
The protocol accepts a tokenized representation of yield-bearing positions (referred to as LYT or SY) and issues two new tokens:

Principal Token (PT): Becomes redeemable for one unit of the underlying asset at maturity.

Yield Token (YT): Allows its holder to claim any yield generated by the position up until the maturity date.

This mechanism separates the principal investment from its yield. The architecture is built on two primary modules:

Vault Module: Manages the economics of PT and YT through stripping (converting SY into PT and YT), staking (allowing YT holders to earn yield), and merging (recombining PT and YT into the original SY). It holds the SY token in escrow.

Market Module: An AMM that handles trading PT with its correlated SY token.

Key User Actions
The platform provides these instruction handlers:

Strip Assets: Convert SY tokens into separate Principal Tokens (PT) and Yield Tokens (YT).

Trade Tokens: Buy YT, sell YT, and trade PT through available market mechanisms.

Provide Liquidity: Deposit PT and SY tokens into the AMM. LP tokens can be deposited to earn farming rewards and yield from the SY tokens they command.

Merge Assets: Combine equal amounts of PT and YT back into SY tokens.

Interfaces and Key Mechanisms
Exponent's Core program integrates with the broader DeFi ecosystem through:

CPI Interface: Enables interaction with various underlying "standard" programs that wrap positions into yield-bearing tokens or standardize existing yield-bearing tokens. This interface ensures compatibility across different types of yield-bearing tokens.

Flash Swaps: Enable liquidity pools to consist only of the base asset (staked as SY) and Principal Tokens (PT). This guarantees no impermanent loss if an LP provider maintains their position until maturity. Flash swaps allow users to buy or sell YT using the SY/PT liquidity pool.

Findings Summary

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

Severity	Discovered	Confirmed	Fixed
Critical	0	-	-
High	0	-	-
Medium	4	4	4
Low	10	10	6
Informational	5	5	2
Total	19	19	12

Medium Severity Issues

M-01 TradePT instruction results in incorrect last_ln_implied_rate

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: exponent_core/src/instructions/market_two/trade_pt.rs	Status: Fixed	

Description: The `MarketTwo::last_ln_implied_rate` should represent the market implied rate immediately after the trade, but the TradePT instruction calculates and sets this value before deducting treasury fees from the SY balance, resulting in an incorrect rate.

```rust
let trade_result = ctx.accounts.market.financials.trade_pt(
    sy_exchange_rate,
    net_trader_pt,
    now,
    is_current_flash_swap,
);
// ... slippage checks and transfers ...
let fee_amount = (trade_result.sy_fee * ctx.accounts.market.fee_treasury_sy_bps as u64) /
10_000;
// ... transfer operations ...
ctx.accounts.market.financials.dec_sy_balance(fee_amount);
```

exponent_core/src/instructions/market_two/trade_pt.rs#L181-L233

The `trade_pt` function calculates the `last_ln_implied_rate` based on the current asset balance, which depends on the SY balance. However, the TradePT instruction handler subsequently deducts treasury fees from the SY balance after the rate has already been computed and stored. This violates the invariant that `last_ln_implied_rate` should accurately reflect the market's state immediately after the complete trade execution, including all fee deductions.

Exploit Scenario:

1. Alice, trader, executes a TradePT instruction to buy PT tokens
2. The TradePT handler processes the trade and calculates treasury fees
3. The `trade_pt` function calculates and sets `last_ln_implied_rate` based on current SY balance
4. Treasury fees are deducted from the market's SY balance via `dec_sy_balance`
5. The stored `last_ln_implied_rate` no longer accurately represents the market state after fee deduction

Recommendations: Move the `last_ln_implied_rate` calculation to occur after all balance adjustments, including treasury fee deductions, to ensure it accurately reflects the market's final state after the complete trade execution.

Customer's response: Fixed in #2488

Fix Review: Fix reviewed

certora exponent

M-02 GetSyState does not ensure Kamino reserve is updated before computing exchange rate

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Files: kamino_lend_standard/src/instructions/read/get_sy_state.rs	Status: Fixed	

Description: The `GetSyState` instruction in Kamino Lend Standard fails to refresh the Kamino reserve before computing the exchange rate, potentially using stale state data that doesn't include recently accrued interest, resulting in an undervalued SY exchange rate.

The instruction computes the SY-to-asset exchange rate directly from the current reserve state without ensuring the reserve has been refreshed to include the latest accrued interest. Kamino reserves accumulate interest over time, and their state must be explicitly updated to reflect current values. When the reserve state is stale, the computed exchange rate will be lower than the actual current rate, creating arbitrage opportunities for users who can exploit the discrepancy.

Exploit Scenario:

1. Alice, user, notices that the Kamino reserve hasn't been updated recently and interest has accrued
2. The Exponent Core program calls `GetSyState` which returns an exchange rate based on stale reserve data
3. The returned rate is lower than the true current exchange rate due to unaccounted accrued interest
4. Alice uses the `Merge` instruction to combine PT and YT tokens into SY at the favorable stale rate

5. Alice provides fewer PT/YT tokens than she should for the equivalent SY amount
6. Alice profits from the exchange rate discrepancy at the expense of other users and the protocol

Alice can also exploit this in AMM trades by selling fewer assets than the actual market value

Recommendations: Ensure the Kamino reserve is refreshed before computing the exchange rate in `GetSyState`. Add a refresh operation or require that the reserve state is current before performing exchange rate calculations to prevent the use of stale data.

Customer's response: Fixed in #2478

Fix Review: Fix reviewed.

M-03 MintSy and RedeemSy instructions fail to refresh treasury emissions causing reward calculation errors

Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Files: sy_standard_*/src/instructions/mint_sy.rs sy_standard_*/src/instructions/redeem_sy.rs	Status: Fixed	

Description: The `MintSy` and `RedeemSy` instructions do not refresh treasury emissions before changing the SY token supply, causing the treasury to earn incorrect reward amounts that can lead to user claim failures due to insufficient funds.

The SY standard programs calculate treasury balance as unstaked SY tokens and use a standard staking rewards distribution process with global and user-specific indices. This system requires updating user rewards on every balance change to maintain accurate reward calculations.

When `MintSy` creates `100` new SY tokens, the unstaked SY balance increases by `100` until the user deposits those tokens. Since treasury emissions are not refreshed during minting, the treasury incorrectly earns rewards for those `100` tokens for the entire period before they were actually minted. Similarly, `RedeemSy` operations can cause the treasury to lose rewards it should have earned. This creates a mismatch between actual reward entitlements and calculated distributions.

Exploit Scenario:

1. Treasury has 10000 unstaked SY tokens
2. Global reward index increases from 1.5 to 2.0
3. Alice, user, calls `MintSy` to create 500 new SY tokens without treasury emission refresh
4. Treasury balance calculation shows 10500 unstaked SY tokens, but treasury index remains at 1.5

5. When treasury claims rewards, it calculates: 10500 * (2.0 - 1.5) = 5250 reward tokens
6. Treasury receives 5250 tokens instead of the correct 5000 tokens it actually earned
7. The extra 250 reward tokens depletes the reward pool meant for legitimate stakers
8. When actual stakers attempt to claim their 250 tokens, transactions fail due to insufficient funds

Recommendations: Add treasury emission refresh calls to both `MintSy` and `RedeemSy` instructions before modifying the SY token supply.

Customer's response: Fixed in #2482

Fix Review: Fix reviewed.

M-04 WithdrawYT does not revert in emergency mode causing inflated emission rewards

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: exponent_core/src/instructions/vault/withdraw_yt.rs	Status: Fixed	

Description: The WithdrawYT instruction fails to check for emergency mode before processing withdrawals, allowing users to earn inflated emission rewards when the SY exchange rate has decreased due to integrated protocol issues.

```rust
    fn total_sy_balance(&self, vault: &Vault) -> u64 {
        self.interest.staged + py_to_sy(vault.final_sy_exchange_rate, self.yt_balance)
    }

    fn earn_emissions(&mut self, vault: &Vault) {
        // TODO - consider negative rates
        let sy_balance = self.total_sy_balance(vault);

        for (index, emission) in vault.emissions.iter().enumerate() {
           [...]
        }
    }
```

exponent_core/src/state/yield_token_position.rs#L101-L117

```rust
pub fn py_to_sy(sy_exchange_rate: Number, amount_py: u64) -> u64 {
    let sy = Number::from_natural_u64(amount_py) / sy_exchange_rate;
    sy.floor_u64()
}
```

exponent_core/src/utils/sy_cpi.rs#L285-L288

Emergency mode occurs when `last_seen_sy_exchange_rate` is less than `all_time_high_sy_exchange_rate`. The `WithdrawYT` instruction updates user emission rewards by calculating their total SY balance, which depends on converting YT balance to SY using the exchange rate.

During emergency mode, the reduced exchange rate causes `py_to_sy` to return inflated SY values. Since emission rewards are calculated based on this SY balance, users receive significantly more emissions than they should, creating a race condition where total claimed emissions exceed available distributions.

Exploit Scenario:

1. Alice, user, has 1000 YT tokens staked when the vault's all-time high exchange rate is 1.5
2. The integrated protocol experiences a hack, causing the SY exchange rate to drop to 1.0 (emergency mode)
3. Alice calls `WithdrawYT` to withdraw her position
4. The instruction calculates her total SY balance: `py_to_sy(1.0, 1000) = 1000 / 1.0 = 1000 SY`.
5. Under normal conditions with rate 1.5: `py_to_sy(1.5, 1000) = 1000 / 1.5 = 666 SY`.
6. Alice's emission rewards are calculated based on 1000 SY instead of 666 SY (50% inflation)
7. If emission index increased by 0.1, Alice receives `1000 * 0.1 = 100` reward tokens instead of `666 * 0.1 = 66`.
8. Multiple users exploit this inflated calculation before emergency measures are implemented
9. The total claimed emissions exceed available emissions, causing legitimate later claims to fail.

Recommendations: Add emergency mode validation to the `WithdrawYT` instruction to prevent updating state when the protocol is in emergency state.

Customer's response: Fixed in #2566

Fix Review: Fix reviewed

Low Severity Issues

L-01 Vault creation can be DoSed through front-running associated token account creation

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: exponent_core/src/instructions/vault/admin/initialize_vault.rs	Status: Acknowledged	

Description: An attacker can prevent vault creation by front-running the `InitializeVault` transaction and creating the associated token account for the SY token escrow before the legitimate transaction executes.

The vault initialization process creates an escrow for SY tokens using the Associated Token Program's `Create` instruction through `create_associated_token_account_2022`.

The `Create` instruction will fail if the associated token account already exists, causing the entire vault creation transaction to revert. Since anyone can create an associated token account for any given account, an attacker can exploit this by monitoring the mempool for `InitializeVault` transactions and front-running them by creating the required associated token account first.

The attack is limited by the fact that the authority address depends on the vault account address, which is only known when the transaction is submitted. This means the attacker must successfully front-run each specific transaction rather than preemptively blocking all possible vaults.

Exploit Scenario:

1. Alice, an admin, submits an `InitializeVault` transaction to create a new vault
2. Eve, an attacker, monitors the mempool and sees Alice's pending transaction

3. Eve extracts the vault address from Alice's transaction and computes the authority address
4. Eve submits a transaction with higher gas fees to create the associated token account for the authority's SY token escrow
5. Eve's transaction executes first, creating the associated token account
6. Alice's `InitializeVault` transaction fails when trying to create the same associated token account
7. Alice must retry with a different vault address to successfully create the vault

Recommendations: Use the Associated Token Program's `CreateIdempotent` instruction instead of `Create` when creating the SY token escrow. This instruction succeeds even if the associated token account already exists, preventing the DoS attack.

Customer's response: Acknowledged

Fix Review: Acknowledged

L-02 User loses emission rewards when new emissions are added to active vaults

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: exponent_core/src/state/yield_token_position.rs	Status: Fixed	

Description: When a new emission is added to an active vault, the `YieldTokenPosition::ensure_trackers` function incorrectly initializes user positions with the vault's `initial_index` instead of zero, causing users to lose rewards that were already earned by the vault.

```rust
fn ensure_trackers(&mut self, vault: &Vault) {
    for (index, emission) in vault.emissions.iter().enumerate() {
        match self.emissions.get_mut(index) {
            Some(_) => {}
            None => {
                self.emissions
                    .push(YieldTokenTracker::new(emission.initial_index, 0));
            }
        }
    }
}
```

exponent_core/src/state/yield_token_position.rs#L120-L130

When `ensure_trackers` creates a new `YieldTokenTracker` for users, it uses `emission.initial_index` as the `last_seen_index`. The `initial_index` represents the current index of the emission when it was added to the vault, which could be non-zero if rewards are earned immediately during the deposit call to the SY program. This means users with existing

YT balances start tracking from the emission's initial index rather than zero, preventing them from claiming rewards that the vault has already accumulated on their behalf. The vault accumulates these rewards during the `AddEmission` process, but users cannot access them due to the incorrect starting index.

Exploit Scenario:

1. Alice, Exponent Core admin, creates a vault for SY token of MarginFi protocol with active positions
2. The vault operates from t1 to tn with users holding YT balances
3. MarginFi protocol begins distributing new reward tokens, making the SY program eligible for immediate rewards
4. Bob, admin of `marginfi_standard` program admin adds the new emission to the SY program, making rewards claimable
5. Alice calls `AddEmission` for the vault, which calls SY program's `deposit` instruction.
    a. `deposit` instruction withdraws rewards from MarginFi incrementing the emission index to `1`.
    b. Vault's position in the SY program earns from these rewards
6. The vault's emission is added with `initial_index = 1` based on the current reward index
7. Alice, user with non-zero YT balance, starts with `last_seen_index = 1` instead of zero
8. Alice cannot claim the rewards already earned by the vault and can only claim future rewards

Recommendations: Use zero as the `last_seen_index` when initializing user emission trackers in ensure_trackers. This ensures users can claim all rewards earned by the vault from the time the emission was added.

While using zero appears to suggest users should earn rewards from when the emission was first added to the SY program, this approach is correct: for emissions added before vault creation, deposit operations refresh tracking indexes to current values before adding YT balance, preventing unearned rewards. For emissions added after vault creation, zero is correct since existing YT balances should earn rewards from when the emission was added to the vault.

Customer's response: Fixed in #2499

Fix Review: Fix reviewed

L-03 User loses emissions on collected interest when rewards are not updated before deduction

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: exponent_core/src/instructions/vault/collect_interest.rs	Status: Fixed	

Description: The `CollectInterest` instruction deducts from `yield_position.interest.staged` without updating emission rewards beforehand, causing users to lose emissions earned by the staged SY between the last reward update and the collection call.

The amount of emissions a user earns depends on `yield_position.interest.staged`. When `CollectInterest` is called, it reduces the staged amount immediately but fails to calculate and update the emission rewards that the staged SY has accumulated since the last interaction. This results in lost emission rewards that should have been attributed to the user's staged SY balance.

The likelihood is low because users can call `StageYield` or perform other operations before collecting interest to update their rewards and avoid the loss.

Exploit Scenario:

1. Alice, user, has staged earned SY from her position
2. Time passes without Alice performing any operations, allowing staged SY to accumulate emission rewards
3. Alice calls `CollectInterest` to withdraw her staged SY
4. The instruction deducts `yield_position.interest.staged` without updating emission rewards first
5. Alice loses the emission rewards that her staged SY earned since the last reward update

Recommendations: Update emission rewards before deducting from `yield_position.interest.staged` in the `CollectInterest` instruction. This ensures users receive all emission rewards earned by their staged SY up to the point of collection.

Customer's response: Fixed in #2500

Fix Review: Fix reviewed

L-04 ModifyFarm instruction computes undistributed tokens incorrectly causing reward claim failures

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: exponent_core/src/instructions/market_two/admin/modify_farm.rs	Status: Fixed	

Description: The `ModifyFarm` instruction incorrectly calculates undistributed tokens by treating the entire `token_farm.amount` as undistributed, when it actually includes tokens that have been distributed but not yet claimed by users, resulting in insufficient tokens for future claims.

```rust
let new_tokens_needed =
    (new_expiration_timestamp as i64 - current_timestamp as i64) * new_rate as i64;

// If the token_farm has less than the required amount, transfer the difference
// Otherwise, transfer surplus to the token_source
if new_tokens_needed > ctx.accounts.token_farm.amount as i64 {
    let required_amount = new_tokens_needed - ctx.accounts.token_farm.amount as i64;
    transfer(ctx.accounts.transfer_ctx(), required_amount as u64)?;
} else {
    let surplus_amount = ctx.accounts.token_farm.amount as i64 - new_tokens_needed;
    transfer(
        ctx.accounts
            .transfer_from_market_ctx()
            .with_signer(&[&ctx.accounts.market.signer_seeds()]),
        surplus_amount as u64,
    )?;
}
```

exponent_core/src/instructions/market_two/admin/modify_farm.rs#L98-L115

The instruction allows changing the token distribution rate and expiry time, calculating token requirements based on the new parameters. However, it incorrectly assumes `token_farm.amount` represents only undistributed tokens, when this balance includes both undistributed tokens and tokens already distributed to users but not yet claimed. This overestimation leads to transferring insufficient tokens in or removing too many tokens out, leaving the account with less than the required amount for future user claims.

Exploit Scenario:

1. Alice, admin, sets up a farm with 1000 tokens at 10 tokens/hour for 100 hours
2. Users stake and earn rewards over 50 hours, with 500 tokens distributed but only 200 claimed
3. The token farm account contains 800 tokens (300 unclaimed + 500 undistributed)
4. Alice calls `ModifyFarm` to change the rate to 5 tokens/hour for remaining 50 hours
5. The instruction calculates new requirement as 250 tokens (5 * 50 hours)
6. Since 800 > 250, it transfers 550 tokens out as "surplus"
7. The account is left with 250 tokens, but 300 tokens are owed to users who haven't claimed
8. When users attempt to claim their 300 tokens, transactions fail due to insufficient balance
9. The farm remains broken until manually refilled with the missing 50 tokens

Recommendations: Compute undistributed tokens as `old_rate * time_till_expiry` instead of using the `token_farm.amount`.

Customer's response: Fixed in #2501

Fix Review: Fix reviewed

L-05 InitSy instruction of kamino_lend_standard assumes all reserves have farms preventing market creation for farmless reserves

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: kamino_lend_standard/src/instructions/admin/init_sy.rs	Status: Acknowledged	

Description: The `kamino_lend_standard` SY program's `InitSy` instruction unconditionally initializes farm-related state for all reserves, making the unsafe assumption that every Kamino reserve will have an associated farm for reward token emissions.

```rust
ctx.accounts.meta.kamino_farm = ctx.accounts.reserve_farm_state.key();
ctx.accounts.meta.kamino_user_metadata = ctx.accounts.user_metadata.key();
ctx.accounts.meta.user_state = ctx.accounts.obligation_farm.key();

// [...]

let init_obligation_farms_for_reserve_cpi_context = CpiContext::new(
    ctx.accounts.kamino_lend_program.to_account_info(),
    init_obligation_farms_for_reserve_cpi_accounts,
);

init_obligation_farms_for_reserve(init_obligation_farms_for_reserve_cpi_context, 0)?;
```

kamino_lend_standard/src/instructions/admin/init_sy.rs#L270-L275

The instruction requires `reserve_farm_state`, `user_metadata`, and `obligation_farm` accounts and calls `init_obligation_farms_for_reserve` for every reserve initialization. However, not all Kamino reserves will have farms that emit additional reward tokens beyond the base lending rewards. This hard requirement prevents Exponent from creating SY tokens and

associated markets for reserves that don't have farms, limiting the protocol's ability to integrate with the full range of Kamino lending markets.

Exploit Scenario:

1. Alice, admin, wants to create an Exponent market for a Kamino USDC reserve
2. The USDC reserve exists and functions normally but has no associated farm for additional rewards
3. Alice attempts to call `InitSy` to create the SY token for this reserve
4. The instruction fails because it cannot find the required `reserve_farm_state` account
5. Alice cannot create the SY token despite the underlying reserve being perfectly functional
6. Exponent loses the opportunity to provide yield trading for this reserve
7. Users miss out on potential yield trading opportunities for farmless but otherwise valid reserves

Recommendations: Make farm initialization conditional by checking if the reserve has an associated farm before attempting to initialize farm-related state.

Customer's response: Acknowledged. Kamino Lend Reserves generally have a pre-initiated Farm associated with them, regardless of whether incentives are currently active. The presence of an unused Farm is expected and does not imply active reward distribution.

Fix Review: Acknowledged.

L-06 Anyone can create SY position for any owner preventing market and vault initialization

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: exponent_core/src/instructions/market_two/admin/market_two_init.rs, exponent_core/src/instructions/vault/admin/initialize_vault.rs	Status: Acknowledged	

Description: The SY program allows anyone to create positions for any owner without authorization checks. Since every Market and Vault in Exponent requires a position in the SY program during initialization, an attacker can front-run these operations by creating the required position first, causing the initialization to fail.

```rust
// Create an account for the Market robot with the SY Program
cpi_init_sy_personal_account(ctx.accounts.sy_program.key(), ctx.remaining_accounts)?;
```

market_two_init.rs#L381-L382

```rust
// Create an account for the vault robot with the SY Program
cpi_init_sy_personal_account(ctx.accounts.sy_program.key(), ctx.remaining_accounts)?;
```

initialize_vault.rs#L305-L306

Both market and vault initialization attempt to create SY positions for their respective accounts. If an attacker monitors pending transactions and creates these positions before the legitimate initialization transactions execute, the initialization will fail due to attempting to create already

existing accounts. The likelihood is low because vault addresses are typically random and market addresses are derived from vaults, making it difficult to predict addresses before transactions are submitted.

Exploit Scenario:

1. Alice, admin, submits a transaction to initialize a new vault with a specific address
2. Eve, attacker, monitors the mempool and extracts the vault address from Alice's pending transaction
3. Eve quickly submits a transaction with higher fees to create an SY position for the vault address
4. Eve's transaction executes first, creating the SY position for the vault
5. Alice's vault initialization transaction fails when attempting to create the already existing SY position
6. Alice must retry with a different vault address to successfully initialize

Recommendations: Modify the SY position creation to use an idempotent operation that succeeds even if the position already exists, or add access controls to the SY program to prevent unauthorized position creation for specific account types like markets and vaults.

Customer's response: Acknowledged

Fix Review: Acknowledged

L-07 GetPosition instruction fails when reallocation is needed due to incorrect payer account

Severity: **Low**	Impact: **Low**	Likelihood: **Medium**
Files: sy_standard_*/src/instructions/read/get_position.rs	Status: Acknowledged	

Description: The `GetPosition` instruction uses the `meta` account as the reallocation payer, but Anchor requires that `realloc::payer` be a system account for rent transfers through the system program. Since `meta` is owned by the SY Program, not the system program, reallocation attempts fail, causing the entire transaction to revert.}

```rust
#[derive(Accounts)]
pub struct GetPosition<'info> {
    #[account(
        mut,
        has_one = jito_vault,
        realloc = Position::size_of(meta.emissions.len()),
        realloc::payer = meta,
        realloc::zero = true,
    )]
    pub position: Account<'info, Position>,

    #[account(
        mut,
        [...]
    )]
    pub meta: Box<Account<'info, SyMeta>>,
```

jito_restaking_standard/src/instructions/read/get_position.rs#L10-L26

Exploit Scenario:

1. Exponent Core Market has a position account created for 2 emissions
2. The SY program adds a third emission, requiring position accounts to expand for additional space
3. Alice, user, calls `WithdrawLp` on the Exponent Core program
4. The `WithdrawLp` instruction calls `GetPositionState` on the market's position via CPI to jito_restaking_standard
5. The `GetPosition` instruction attempts to reallocate the position account with additional space
6. Anchor tries to transfer rent from the `meta` account using the system program
7. The transfer fails because `meta` is owned by the SY Program, not the system program
8. The entire operation reverts, preventing Alice from withdrawing LP tokens from her market

Recommendations: Add a separate system account as the payer for reallocation operations.

Customer's response: Acknowledged. Reallocation of the position account fails due to the absence of a payer. This behavior is understood and there is no intention to introduce a payer field. The intended workaround is to transfer sufficient lamports to the position account prior to reallocation.

Fix Review: Acknowledged

L-08 Liquidity pool receives less fee due to incorrect formula in asset fee calculation

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: libraries/time_curve/src/math.rs	Status: Fixed	

Description: The `asset_fee` function in the time_curve module uses an incorrect formula when calculating fees for PT purchases, resulting in the liquidity pool receiving less fees than intended.

```rust
pub fn asset_fee<N: Num>(net_trader_asset: N, fee_rate: N) -> N {
    assert!(fee_rate >= N::one());
    let is_sell_pt = net_trader_asset > N::zero();
    if is_sell_pt {
        // selling PT to buy asset
        net_trader_asset * (fee_rate - N::one())
    } else {
        // buying PT and spending asset
        -net_trader_asset * (fee_rate - N::one()) / fee_rate
    }
}
```

libraries/time_curve/src/math.rs#L233-L243

When buying PT (else branch), the `net_trader_asset` represents the amount before fee addition, but the fee rate represents the percentage on the gross amount. The current implementation uses the formula `net_trader_asset * (fee_rate - 1) / fee_rate`, while the mathematically correct formula should be `net_trader_asset * (fee_rate - 1) / (2 - fee_rate)`. This discrepancy causes users to pay lower fees than intended, reducing the liquidity pool's fee revenue.

Exploit Scenario:

1. Alice, trader, wants to buy PT by spending assets with fee_rate = 1.05
2. Alice's net_trader_asset = -1500
3. Using current incorrect formula: computed_fee = 1500 * (1.05 - 1) / 1.05 = 71.43
4. Cross-check shows gross_amount = 1500 + 71.43 = 1571.43, fee_from_gross = 1571.43 * 0.05 = 78.57
5. The calculated fee (71.43) doesn't match the expected fee from gross amount (78.57)
6. Using correct formula: fee = 1500 * (1.05 - 1) / (2 - 1.05) = 78.94
7. Cross-check confirms: gross_amount = 1578.94, fee_from_gross = 78.94
8. The liquidity pool loses approximately 7.5 assets on the purchase of 1500 PT

Recommendations: Update the asset fee calculation formula in the else branch to use the mathematically correct formula.

Customer's response: Fixed in #2477

Fix Review: Fix Reviewed

L-09 Overly strict constraints cause BuyYT instruction to fail with minor exchange rate fluctuations

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: exponent_core/src/instructions/market_two/buy_yt.rs	Status: Fixed	

Description: The BuyYT instruction includes excessive slippage constraints that cause transactions to fail with slight exchange rate changes between transaction preparation and execution.

```rust
assert!(
    received_sy <= sy_to_borrow + 2,
    "received SY must be less than or equal to borrowed SY (with a buffer of 2)"
);

assert!(
    received_sy >= sy_to_borrow,
    "received SY must be greater than or equal to borrowed SY"
);
```

exponent_core/src/instructions/market_two/buy_yt.rs#L278-L286

The BuyYT instruction involves selling a user-specified `yt_out` amount of PT tokens, with constraints that the received SY amount must be in the range `[sy_to_borrow, sy_to_borrow + 2]`. The `sy_to_borrow` value depends on user arguments `max_sy_in` and `yt_out` through the calculation `sy_to_borrow = sy_to_strip - max_sy_in` where `sy_to_strip = floor(yt_out / sy_exchange_rate) + 1`. This tight constraint window of only 2 units

causes transactions to fail with minor exchange rate fluctuations that occur between transaction preparation and execution.

Exploit Scenario:

1. Alice, user, prepares a BuyYT transaction with `yt_out` and `max_sy_in` arguments based on current exchange rate
2. Alice calculates that selling PT should return approximately `sy_to_borrow` SY tokens
3. Between transaction preparation and execution, the exchange rate shifts slightly due to other market activity
4. Alice's transaction executes but receives `sy_to_borrow + 3` SY tokens due to the rate change
5. The transaction fails on the upper bound assertion despite being economically favorable to Alice
6. Alice must recalculate and resubmit the transaction with updated parameters

Recommendations: Remove the upper bound constraint `received_sy <= sy_to_borrow + 2` and only enforce the lower bound `received_sy >= sy_to_borrow` for slippage protection. The upper bound constraint provides no meaningful protection and causes unnecessary transaction failures when users receive more favorable rates than expected.

Customer's response: Fixed in #2489

Fix Review: Fix reviewed

L-10 Incomplete instruction data causes marginfi deposit CPI to fail

Severity: **Low**	Impact: **Low**	Likelihood: **High**
Files: libraries/marginfi_cpi/src/lib.rs	Status: Fixed	

Description: The `deposit_to_lending_account` function only provides the `amount` argument when calling marginfi's `lending_account_deposit` instruction, but the target instruction expects two arguments: `amount` and `deposit_up_to_limit`, causing deserialization failures and preventing SY token minting for marginfi positions.

```rust
let mut instruction_data: Vec<u8> = vec![171, 94, 235, 103, 82, 64, 212, 140];
// @issue lending_account_deposit has two arguments amount, deposit_up_to_limit
instruction_data.extend_from_slice(&amount.to_le_bytes());

let instruction = anchor_lang::solana_program::instruction::Instruction {
    program_id: ID,
    accounts: account_metas,
    data: instruction_data,
};
```

libraries/marginfi_cpi/src/lib.rs#L97-L104

The marginfi V2 `lending_account_deposit` function requires both `amount: u64` and `deposit_up_to_limit: Option<bool>` parameters, but the marginfi_standard program only serializes the amount value into the instruction data. When marginfi attempts to deserialize the instruction data, it fails due to the missing second argument, causing the entire transaction to revert and preventing users from minting SY tokens backed by marginfi lending positions.

Exploit Scenario:

1. Alice, user, attempts to mint SY tokens through the marginfi_standard program
2. The marginfi_standard program calls `deposit_to_lending_account` with only the deposit amount
3. Marginfi attempts to deserialize expecting both `amount` and `deposit_up_to_limit` arguments
4. Deserialization fails due to insufficient instruction data, reverting the transaction
5. All marginfi-based SY token functionality becomes unusable

Recommendations: Add the missing `deposit_up_to_limit` argument to the instruction data serialization.

Customer's response: Fixed in #2490

Fix Review: Fix reviewed

Informational Severity Issues

I-01. Users lose yield on earned SY when yield is staged after vault maturity

Description: The `YieldTokenPosition::earn_sy_interest` function incorrectly scales down earned SY when users stage their yield after vault maturity, causing users to lose yield that should rightfully belong to them.

```rust
fn earn_sy_interest(&mut self, vault: &mut Vault) {
    [...]
    let rate = vault.final_sy_exchange_rate;
    let sy_earned = self.calc_earned_sy(rate);

    // if the current rate is higher than the final rate, scale down the earned SY
    let sy_earned = scale_sy_to_current_rate(sy_earned, rate,
vault.last_seen_sy_exchange_rate);

    self.interest.inc_staged(sy_earned);
    [...]
}
```

exponent_core/src/state/yield_token_position.rs#L67-L85

When users call `StageYield` after vault maturity, the function calculates earned SY based on the vault's final exchange rate and scales down the SY to have the same value as immediately after vault maturity.

Scaling down is incorrect after vault has matured as it removes the yield generated by the earned SY from vault maturity until the user's interaction. This creates an unfair scenario where users who stage their yield earlier receive more rewards than users who stage later, despite having identical positions and contributions during the vault's active period.

Exploit Scenario:

1. Alice, user, deposits 100 YT into her yield position at t1 when vault SYIndex = 1.0
2. Bob, user, deposits 100 YT into his yield position at t1 with the same conditions
3. Vault matures at t2 with final_sy_exchange_rate = 2.0, both users' last_seen_index = 1.0

4. Alice calls `StageYield` immediately at t2, earning 50 SY without scaling
5. By t3, vault SYIndex increases to 2.5
6. Bob calls `StageYield` at t3, calculating 50 SY but scaled down to 40 SY due to the higher current rate
7. Alice receives 50 SY while Bob receives only 40 SY despite identical contributions and timing
8. Bob loses 10 SY worth of yield that his earned SY should have generated from t2 to t3

Recommendations: Remove the scaling operation from `earn_sy_interest` and allow users to receive their full calculated SY earnings.

Customer's response: Acknowledged. At maturity, interest is expected to be locked in terms of the base asset. Any interest accrued on already staged SY after maturity is not claimable by the user and is directed to the market's treasury. This behavior is intended and aligns with the protocol's design.

Fix Review: Acknowledged.

I-02. SyStandard programs lack instructions to update configuration parameters

Description: The SyStandard programs do not provide administrative instructions to update key configuration parameters such as `min_mint_size`, `min_redeem_size`, and `max_sy_supply` after initialization. These parameters are set during the `InitSy` instruction and become immutable for the lifetime of the SY token.

For example, the marginfi_standard program sets `max_sy_supply` to `u64::MAX` by default during initialization, but there is no mechanism to adjust this limit if needed for risk management or operational requirements. Similarly, minimum mint and redeem sizes cannot be updated to reflect changing market conditions or protocol requirements.

Recommendation: Add administrative instructions to allow updating these configuration parameters post-deployment.

Customer's response: Acknowledged. There is currently no need to adjust the SY parameters. The preference is to keep these parameters immutable to preserve protocol stability and reduce configuration surface.

Fix Review: Acknowledged

I-03. Lack of global emergency switch to pause all operations simultaneously

Description: The Exponent Core program lacks a global emergency switch that can pause operations across all vaults and markets simultaneously. The current implementation only allows pausing individual markets and vaults one at a time.

Given the protocol's architecture with multiple yield tokens (one per integrated protocol and asset combination) and corresponding markets and vaults for each expiry date, the total number of markets and vaults grows exponentially. During a security incident or attack, administrators would need to manually pause each market and vault individually, which is time-consuming and impractical when rapid response is critical.

Recommendation: Implement a global emergency switch for the Exponent Core program that can pause all operations across markets and vaults with a single instruction.

Customer's response: Acknowledged. A global emergency pause mechanism is not implemented. Instead, specific markets can be paused in batches as needed.

Fix Review: Acknowledged

I-04. SyMeta account size calculation includes excess space locking unnecessary rent

Description: The `size_of` function for the SyMeta account of the `kamino_lend_standard` program calculates an account size that exceeds the actual required space, resulting in paying excess lamports for rent that remain locked unnecessarily.

Recommendation: Review and recalculate the actual space requirements for the SyMeta struct by comparing the `size_of` calculation against the real struct layout.

Customer's response: Fixed in #2502

Fix Review: Fix reviewed

I-05. BuyYT instruction charges less fee decreasing the profit for LP providers

Description: The BuyYT instruction combines stripping SY and selling PT to acquire only YT tokens, but charges fees based on YT value instead of PT value resulting in significantly lower fees for the AMM and reduced profits for LP providers.

```rust
let fee = if is_current_flash_swap {
    // 1 / er represents the PT price in asset terms
    // Therefore, (1 - (1/er)) represents the YT price in asset terms
    let yt_value = (N::one() - N::one() / er) * net_trader_pt.abs();
    asset_fee(yt_value, fee_rate)
} else {
    asset_fee(pre_fee_net_trader_asset, fee_rate)
};
```

time_curve/src/math.rs#L285-L289

When BuyYT executes, it sells PT using the TradePT instruction with `is_current_flash_swap = true`, causing the trade to charge fees on the YT value rather than the full PT value. Since YT value represents only the yield component and is typically much lower than PT value (which represents the principal asset), this results in substantially reduced fee collection.

The AMM should charge higher fees for providing flash swap facilities, but instead charges less, making BuyYT more cost-effective than standard trading even for users who don't require SY loans

Exploit Scenario:

1. Alice, user, wants to buy 1000 YT tokens with 1000 SY, sy_exchange_rate = 1.5, AMM rate = 1.2, fee_rate = 1.05
2. **Scenario 1 (Standard approach):** Alice borrows 500 SY, strips 1500 SY for 1000 PT + 1000 YT, sells 1000 PT via TradePT
3. TradePT charges fee on full PT value: fee = 833 * (1.05 - 1) = 41.65, Alice gets 791.35 assets = 527.56 SY
4. Alice repays 505 SY, ending with 22.56 SY and 1000 YT

5. **Scenario 2 (BuyYT):** Alice calls BuyYT with same parameters
6. BuyYT borrows 500 SY, strips to get PT + YT, sells PT with `is_current_flash_swap = true`
7. Fee calculated on YT value: yt_value = 166.66, fee = 166.66 * 0.05 = 8.33
8. Alice gets 824.67 assets = 549.78 SY, repays 500 SY, ending with 49.78 SY and 1000 YT
9. Alice pays significantly lower fees (8.33 vs 41.65) and retains more SY (49.78 vs 22.56)

Recommendations: Modify the fee calculation for flash swaps to charge the standard fee on the full PT value being traded plus an additional fee for providing the flash swap feature. This ensures LP providers receive appropriate compensation for both the liquidity provided and the additional service of enabling flash swaps, maintaining proper economic incentives.

Customer's response: Partially fixed. The current behaviour is expected, as charging fees based on the amount of PT traded would incur a substantial amount of fees as a market approaches maturity. We agree that the fee rate for YT trades should be adjustable though, with a multiplier based on how much the market's administrator wants to incentivize YT trading. Added flash swap fee multiplier in #2485

Fix Review: Acknowledged and Fix reviewed

Suggested Code Improvements

Remove Incorrect Comments

- **Remove incorrect comment about f64 implementation safety since it's actually used in production code.** The comment states f64 is only for tests, but `trade_pt` uses f64 in production calls to `exponent_time_curve::math::trade`.

```rust
 // note - the f64 implementation is not safe for financial calculations
// this is just used for tests
impl Num for f64 {
```

libraries/time_curve/src/num.rs#L65-L66

- **Correct comment about `delta_time_farm` function to reflect actual behavior.** The function returns zero when `last_seen_timestamp >= expiry`, not when current timestamp is greater than expiry.

```rust
// if current timestamp is greater than the expiry, this function returns 0
let time_delta = delta_time_farm(
```

programs/exponent_core/src/state/market_two.rs#L865-L866

- **Remove incorrect comment about Token2022 usage since the Token program is actually used.**

```rust
/// Use Token2022 as the implementation for PT & SY & LP tokens
pub token_program: Program<'info, Token>,
```

exponent_core/src/instructions/market_two/admin/market_two_init.rs#L92-L94

- **Remove incorrect comment about virtual liquidity mutation.**

```rust
// mutate the market financials, increasing the SY balance
```

```
// the trade was made with "virtual liquidity", and the purchase of SY must be "returned" to
the market
// ctx.accounts.market.financials.inc_sy_balance(sy_to_borrow);
```

exponent_core/src/instructions/market_two/buy_yt.rs#L293-L295

- **Correct comment for `WithdrawYt` struct since it describes deposit functionality.**

```rust
/// Deposit YT into escrow, increasing the balance stored in owner's YieldTokenPosition
///
/// YT does not actively earn interest (SY) and emissions unless it is deposited into the
program
/// Upon depositing, the owner's earned interest (SY) and emissions will be staged
///
#[event_cpi]
#[derive(Accounts)]
pub struct WithdrawYt<'info> {
```

exponent_core/src/instructions/vault/withdraw_yt.rs#L13-L19

- **Update comment for marginfi_standard `min_mint_size` field to accurately describe its purpose.**

```rust
/// Minimum size of SY tokens that can be made from minting
/// This is to prevent dust from being minted, or from rounding errors
pub min_mint_size: u64,
```

marginfi_standard/src/state/sy_meta.rs#L24-L26

- **Add missing comment to explain pre-fee amount handling in asset fee calculation.**

```rust
} else {
    // buying PT and spending asset
    -net_trader_asset * (fee_rate - N::one()) / fee_rate
```

libraries/time_curve/src/math.rs#L240-L241

Remove Unused Code and Comments

- **Remove commented-out `uncollected_sy` field definition and its description.** The field is no longer used in the codebase.

```rust
/// SY that has been staged for collection, but not yet collected
/// It is strictly greater-than-or-equal to the treasury_sy
/// And strictly less than or equal to the total_sy_in_escrow
// pub uncollected_sy: u64,
```

exponent_core/src/state/vault.rs#L84-L87

- **Remove unused `Vault::can_stage_sy_interest` function.** The function is not referenced anywhere in the codebase.

```rust
/// Can stage interest only if the vault is active and not in emergency mode
pub fn can_stage_sy_interest(&self, current_ts: u32) -> bool {
    self.is_active(current_ts) && !self.is_in_emergency_mode()
}
```

exponent_core/src/state/vault.rs#L126-L129

- **Remove unused `Vault::size_of` function.** The function is not used and can be replaced with standard serialization methods if needed.

```rust
pub fn size_of(&self) -> usize {
    self.try_to_vec().unwrap().len()
}
```

exponent_core/src/state/vault.rs#L140-L143

- **Remove unused `SurplusYield` struct.** The struct is defined but never used in the codebase.

```rust
pub struct SurplusYield {
    pub sy_surplus: u64,
```

```
    pub emission_surpluses: Vec<u64>,
}
```

programs/exponent_core/src/state/vault.rs#L505-L508

- **Remove unused `rent` account from marginfi `init_sy` instruction.** The account is declared but never used.

```Rust
pub rent: Sysvar<'info, Rent>,
```

marginfi_standard/src/instructions/admin/init_sy.rs#L114

- **Remove unnecessary signer seeds from SY CPI call and change `invoke_signed` to `invoke`.** The `seeds` parameter is not required for this operation.

```Rust
pub fn cpi_get_position<'i>(
    [...]
) -> Result<PositionState> {
    let mut data: Vec<u8> = vec![];

    let discriminator = [10];
    data.extend_from_slice(discriminator.as_slice());

    invoke_signed(
        &Instruction {
            program_id: sy_program,
            accounts: account_metas,
            data,
        },
        &account_infos,
        seeds,
    )?;
```

exponent_core/src/utils/sy_cpi.rs#L184-L203

Improve Function Names and Constants

- **Rename `LpFarm::get_farm_emission_index` to avoid confusion with reward indexes.** The function returns array position, not reward index values.

```rust
pub fn get_farm_emission_index(&self, mint: Pubkey) -> Option<usize> {
    self.farm_emissions
        .iter()
        .position(|emission| emission.mint == mint)
}
```

exponent_core/src/state/market_two.rs#L852-L856

- **Use `AUTHORITY` constant in `Vault::signer_seeds` instead of hardcoded string.** This improves maintainability and reduces magic strings.

```rust
pub fn signer_seeds(&self) -> [&[u8]; 3] {
    [
        b"authority".as_ref(),
        self.signer_seed.as_ref(),
        &self.signer_bump,
    ]
}
```

exponent_core/src/state/vault.rs#L132-L138

- **Change the field name `token_22_program` to better reflect its actual definition.**

```rust
pub token_22_program: Program<'info, Token>,
```

programs/kamino_lend_standard/src/instructions/admin/init_sy.rs#L117

Replace Functions with More Appropriate Alternatives

- **Replace `Vault::collect_treasury_interest` with `dec_treasury_sy` for consistency.** Using the standard decrement function maintains consistency across the codebase.

```rust
pub fn collect_treasury_interest(&mut self, amount: u64) {
    self.treasury_sy = self.treasury_sy.checked_sub(amount).unwrap();
}
```

exponent_core/src/state/vault.rs#L150-L152

- **Replace `YieldTokenTracker::collect` with `dec_staged` for consistency.** Using the standard decrement function maintains consistency across the codebase.

```rust
pub fn collect(&mut self, amount: u64) {
    self.staged = self.staged.checked_sub(amount).unwrap();
}
```

exponent_core/src/state/yield_token_position.rs#L190-L192

Optimize Code Patterns

- **Replace if-else branch in `MarketTwo::sec_remaining` with `saturating_sub`.** This is more concise and handles underflow automatically.

```rust
fn sec_remaining(&self, now: u64) -> u64 {
    if now > self.expiration_ts {
        0
    } else {
        self.expiration_ts - now
    }
}
```

exponent_core/src/state/market_two.rs#L377-L383

- **Replace `for` loop with simple `while` loop in `PersonalYieldTrackers::ensure_trackers`.**

```rust
fn ensure_trackers(&mut self, emission_indexes: &EmissionIndexes) {
    for (pos, _) in emission_indexes.iter().enumerate() {
        match self.trackers.get(pos) {
            Some(_) => {}
            None => {
                self.trackers.push(PersonalYieldTracker {
                    last_seen_index: Number::ZERO,
                    staged: 0,
                });
```

```
            }
        }
    }
}
```

exponent_core/src/state/personal_yield_tracker.rs#L33-L45

- **Add a function that performs ceil division instead of adding 1 to floor division in BuyYT instruction.**

```Rust
let sy_to_strip = py_to_sy(sy_exchange_rate, yt_out) + 1
```

exponent_core/src/instructions/market_two/buy_yt.rs#L200

- **Simplify boolean expression by removing unnecessary comparison.** The `!= false` comparison is redundant.

```Rust
if self.get_mfi_account().lending_account.balances[0].active != false {
    self.withdraw_emissions()?;
}
```

programs/marginfi_standard/src/utils.rs#L133-L135

- **Consider adding `to_u64_ceil()` to Num trait for cleaner ceiling operations.** This would improve code readability for ceiling calculations.

```Rust
let sy_in = ((market_total_sy * lp_tokens_out + market_total_lp - N::one())
    / market_total_lp)
    .to_u64();
```

time_curve/src/math.rs#L150-L152

Add Missing Validations and Error Handling

- **Add maximum BPS validation in `ChangeVaultBpsFee` to prevent invalid fee values.** Fee values should not exceed 10,000 basis points (100%).

```rust
AdminAction::ChangeVaultBpsFee(new_fee) => {
    ctx.accounts
        .admin_state
        .principles
        .exponent_core
        .is_admin(ctx.accounts.signer.key)?;

    vault.interest_bps_fee = new_fee;
}
```

exponent_core/src/instructions/vault/admin/modify_vault_setting.rs#L70-L77

- **Replace TODO statements with proper error codes in liquidity operations.** Proper error handling improves user experience and debugging.

```rust
if r.lp_out < min_lp_out {
    todo!("ERROR CODE: min_lp_out not met");
}
```

exponent_core/src/instructions/market_two/deposit_liquidity.rs#L160-L162

```rust
if r.sy_out < min_sy_out {
    todo!("ERROR CODE: min_sy_out not met");
}
```

exponent_core/src/instructions/market_two/withdraw_liquidity.rs#L167-L169

- **Add market activity validation in BuyYT and SellYT instructions.** Although checked indirectly via CPI to TradePt, explicit validation improves code clarity.

```rust
pub fn validate(&self) -> Result<()> {
    require!(
        self.market.check_status_flags(STATUS_CAN_BUY_YT),
        ExponentCoreError::BuyingYtDisabled
    );
```

exponent_core/src/instructions/market_two/buy_yt.rs#L166-L170

- **Ensure the market is active in `DepositLiquidity` instruction validation.**

- **Add emission mint validation in `AddEmission` instruction of `kamino_lend_standard`.** Ensure emission mints are in the same order as reward infos of the reserve.

- **Add token mint validation for emission token accounts.** Ensure the token account mint matches the emission mint.

```rust
/// Assert that the new token account is owned by the vault
#[account(
    token::authority = vault.authority,
)]
pub robot_token_account: InterfaceAccount<'info, TokenAccount>,
```

exponent_core/src/instructions/vault/admin/add_emission.rs#L34-L39

- **Add slippage validation in SellPT wrapper instruction.** Include minimum base amount check after redeeming SY.

```rust
let redeem_sy_return_data = sy_cpi::cpi_redeem_sy(
    ctx.accounts.sy_program.key(),
    trade_pt_return_data.net_trader_sy as u64,
    redeem_base_accounts,
    redeem_base_accounts.to_vec().to_account_metas(None),
)?;

// @cq add a redeem_sy_return_datebase_out_amount >= min_base_amount check here
```

exponent_core/src/instructions/wrappers/sell_pt.rs#L51-L56

- **Fix incorrect admin validation in perena_standard init_sy instruction.** Should validate against perena_standard admin, not jito_restaking admin.

```rust
pub fn validate(&self) -> Result<()> {
    self.admin_state
```

```
        .principles
        .jito_restaking
        .is_admin(&self.admin.key)?;
```

perena_standard/src/instructions/admin/init_sy.rs#L170-L175

Improve Code Organization and Readability

- **Reduce nested function calls in LP token transfer operations.** Consider following the rule of 3-4 nesting levels maximum for better readability.

```rust
fn do_transfer_lp_in(&mut self, amount: u64) -> Result<()> {
    #[allow(deprecated)]
    token_2022::transfer(self.transfer_lp_in_context(), amount)?;

    self.market.lp_escrow_amount = self.market.lp_escrow_amount.checked_add(amount).unwrap();

    Ok(())
}
```

exponent_core/src/instructions/market_two/deposit_lp.rs#L56-L78

- **Move log message outside loop to avoid cluttering logs.** Emit the message once instead of for each iteration.

```rust
for (index, escrow_account) in escrow_accounts.into_iter().enumerate() {
    let i = 5 * index;
    let target_token_account = remaining_accounts[i].key;

    assert_eq!(
        target_token_account, escrow_account,
        "Target token account mismatch"
    );

    // @cq emit the msg outside the loop to avoid cluttering the logs
    msg!(
        "In Process Reward Emissions: user_state key: {:?}",
        user_state.key()
```

```
    );
```

kamino_lend_standard/src/utils.rs#L90-L93

- **Optimize claim logic in CollectEmission instruction.** Claim all claimable amount into escrow and use conditional checks to avoid unnecessary SY program calls when escrow has sufficient balance.

```Rust
cpi_claim_emission(
    ctx.accounts.sy_program.key(),
    amount_to_send,
    ctx.remaining_accounts,
    to_account_metas(
        &ctx.accounts.vault.cpi_accounts.claim_emission[index as usize],
        &lookup_table,
    ),
    signer_seeds,
)?;
```

exponent_core/src/instructions/vault/collect_emission.rs#L99-L107

- **Remove redundant fee capping in CollectInterest instruction.** The cap is only needed if interest_bps_fee exceeds 10,000.

```Rust
// cap the fee at the amount_sy
let fee_sy = fee_sy.min(amount_sy as u128) as u64;
```

exponent_core/src/instructions/vault/collect_interest.rs#L172-L173

- **Reconsider redundant assert statements in BuyYT instruction.** The assertions may be unnecessary given the mathematical constraints.

```Rust
assert!(
    sy_to_strip > max_sy_in,
    "sy_to_strip must be greater than max_sy_in"
);
```

```
let sy_to_borrow = sy_to_strip - max_sy_in;

// Check borrow amount is less than target strip amount
assert!(
    sy_to_borrow < sy_to_strip,
    "sy_to_borrow must be less than sy_to_strip"
);
```

exponent_core/src/instructions/market_two/buy_yt.rs#L202-L213

- **Use `pt_out` instead of `yt_out` for PT sell amount calculation.** `pt_out` represents the exact amount received and need to be sold.

```
Rust
let pt_sell: i64 = yt_out.try_into().unwrap();
```

exponent_core/src/instructions/market_two/buy_yt.rs#L256-L272

Missing Instruction Requirements

- **Ensure vault is active in InitializeYieldPosition instruction.** Add validation to prevent position creation for inactive vaults.
- **Call `ensure_yield_trackers` on the new position in InitializeYieldPosition instruction.** Initialize emissions tracking for the new position.
- **Ensure the market is active in InitLpPosition instruction.** Add validation to prevent LP position creation for inactive markets.

Authorization Inconsistencies

- **Use `collect_treasury` admin instead of `kamino_lend_standard` admin for treasury emission collection.** The treasury collection should be governed by treasury-specific permissions.
- **Make kamino_lend_standard program deposit permissions consistent with other standard programs.** Allow anyone to deposit, not just the position owner, for consistency across standard programs.

Disclaimer

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

About Certora

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

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