# Exponent Finance

Security Assessment

October 16th, 2024 — Prepared by OtterSec

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Akash Gurugunti sud0u53r.ak@osec.io

Robert Chen r@osec.io

# Table of Contents

# 01 — Executive Summary

## Overview

Exponent Finance engaged OtterSec to assess the `exponent-core` program. This assessment was conducted between August 24th and October 9th, 2024. For more information on our auditing methodology, refer to Appendix B.

## Key Findings

We produced 22 findings throughout this audit engagement.

In particular, we identified a critical vulnerability where the interest collection instructions fail to verify that the provided SY program matches the expected program in the vault (OS-EXF-ADV-00) and another issue concerning the lack of proper access control checks in the instructions for adding a farm and sending farm tokens (OS-EXF-ADV-08). We also highlighted several mathematical inconsistencies arising from the use of incorrect operations (OS-EXF-ADV-05), as well as several cases of improper reward distribution due to the utilization of outdated values (OS-EXF-ADV-03).

Furthermore, when selling principal tokens, the treasury fee is transferred from the escrow to the treasury, but the corresponding amount is not deducted from the market's balance of SY tokens (OS-EXF-ADV-01). Additionally, the floor operation returns an incorrect value (OS-EXF-ADV-12).

We also made recommendations to ensure adherence to coding best practices (OS-EXF-SUG-05) and suggested removing unused and redundant code within the system for increased readability (OS-EXF-SUG-06). We further advised incorporating additional checks within the codebase for improved robustness and security (OS-EXF-SUG-01) and modifying the codebase for enhanced functionality, efficiency, and maintainability (OS-EXF-SUG-04).

# 02 — Scope

The source code was delivered to us in a Git repository at https://github.com/exponent-finance/exponent-core. This audit was performed against commit a9d3a6b.

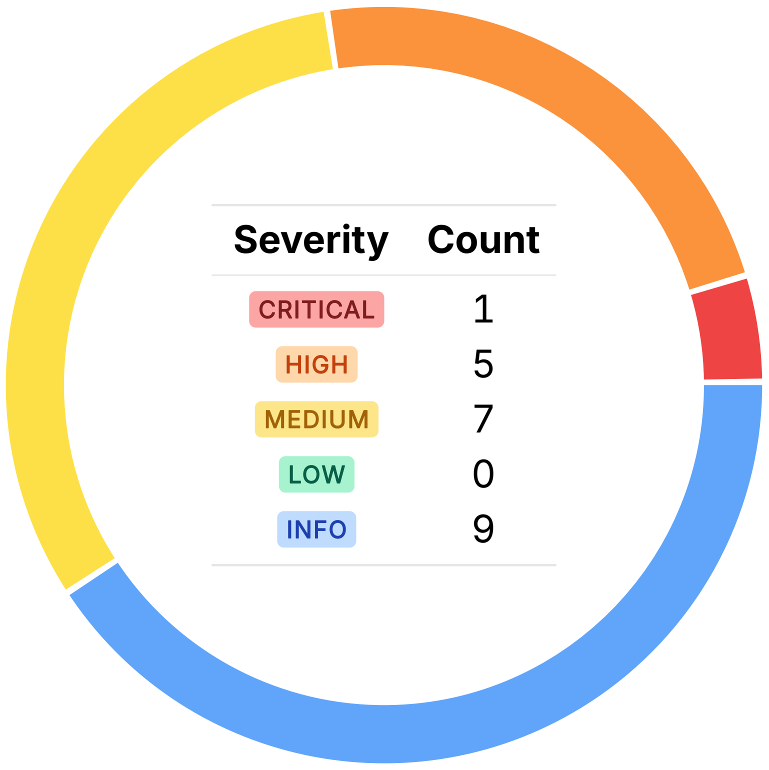
A brief description of the program is as follows:

| Name | Description |
|---|---|
| exponent-core | A protocol that allows users to strategically trade tokenized future yields by splitting assets into Principal and Yield Tokens for flexible yield management. It also incorporates farm emissions to incentivize user participation and enhance liquidity. |

# 03 — Findings

Overall, we reported 22 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



| Severity | Count |
|---|---|
| CRITICAL | 1 |
| HIGH | 5 |
| MEDIUM | 7 |
| LOW | 0 |
| INFO | 9 |

# 04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

| ID | Severity | Status | Description |
|---|---|---|---|
| OS-EXF-ADV-00 | CRITICAL | RESOLVED ✓ | `CollectTreasuryInterest` and `CollectInterest` do not verify that the `sy_program` provided matches the expected program in the vault. |
| OS-EXF-ADV-01 | HIGH | RESOLVED ✓ | In the `TradePt` instruction, when selling `PT`, the treasury fee is transferred from the escrow to the treasury, but the corresponding amount is not deducted from the market's `SY` balance. |
| OS-EXF-ADV-02 | HIGH | RESOLVED ✓ | `market.lp_escrow_amount` is not decremented during LP token transfers in `do_transfer_lp_out`, resulting in potential mismanagement of escrow balances and inaccurate accounting. |
| OS-EXF-ADV-03 | HIGH | RESOLVED ✓ | The failure to update changes, such as emission indexes and `last_seen_staged`, results in the utilization of outdated values before retrieving reward amounts. |
| OS-EXF-ADV-04 | HIGH | RESOLVED ✓ | `buy_yt` lacks a critical `do_deposit_sy` `CPI` call after `do_repay_sy`, resulting in `SY` tokens remaining locked in the `token_sy_escrow` account. |
| OS-EXF-ADV-05 | HIGH | RESOLVED ✓ | There are several mathematical inconsistencies as a result of the utilization of incorrect operations. |

| | | | |
|---|---|---|---|
| OS-EXF-ADV-06 | MEDIUM | RESOLVED ✓ | `increase_share_indexes` and `update_emissions_from_position_state` incorrectly distribute rewards across the entire LP supply rather than just for the LP tokens deposited in active LP positions. |
| OS-EXF-ADV-07 | MEDIUM | RESOLVED ✓ | In scenarios where the exchange rate decreases, `withdraw_yt` and `merge` lack safeguards against increasing the `sy_for_pt` amount, enabling attackers to withdraw large amounts of `SY`. |
| OS-EXF-ADV-08 | MEDIUM | RESOLVED ✓ | `add_farm` and `send_farm_token` lack proper access control checks, and the `token_farm` account is assigned an incorrect authority. |
| OS-EXF-ADV-09 | MEDIUM | RESOLVED ✓ | The `AddMarketEmission` instruction reuses `market.cpi_accounts` to reallocate the market size instead of using the provided `cpi_accounts` from the input. |
| OS-EXF-ADV-10 | MEDIUM | RESOLVED ✓ | `modify_market_setting` instruction fails to update the share indexes with `increase_share_indexes` before adjusting the `new_rate` for farm emissions, resulting in inaccurate reward distribution. |
| OS-EXF-ADV-11 | MEDIUM | RESOLVED ✓ | In the vault account, `treasury` and `treasury_emission`, which are accumulated in `increase_lambo_fund`, are neither retrieved nor utilized anywhere. |
| OS-EXF-ADV-12 | MEDIUM | RESOLVED ✓ | `floor_u128` in `precise_number` does not correctly apply the floor operation, returning incorrect values. |

## Lack of SY Program Verification CRITICAL OS-EXF-ADV-00

### Description

In the `CollectInterest` and `CollectTreasuryInterest` (shown below) instructions, the `sy_program` is not verified against the vault. The `sy_program` is intended to manage the `SY` tokens and should ideally be verified to ensure that it is the correct and authorized program for handling these tokens. Thus, it may be possible to replace the `sy_program` with a malicious program to potentially redirect the `SY` tokens.

>_ *vault/admin/treasury/collect_treasury_interest.rs* RUST

```rust
#[derive(Accounts)]
pub struct CollectTreasuryInterest<'info> {
    [...]
    /// CHECK: constrained by vault
    pub sy_program: UncheckedAccount<'info>,
}
```

Specifically, it may affect `YT` holders, resulting in discrepancies when they try to collect the interest they earned from their `YT` holdings, and administrators when they try to withdraw `SY` tokens that have accumulated as interest in a vault.

### Remediation

Ensure that the `sy_program` is appropriately validated in `CollectInterest` and `CollectTreasuryInterest`.

### Patch

Resolved in PR#520.

## SY Balance Fee Accounting Error HIGH OS-EXF-ADV-01

### Description

In the `TradePt` instruction in `exponent_core`, when the user sells `PT` (Principal Tokens), they receive `SY` tokens (Stake Yield tokens) in return. A portion of these `SY` tokens is taken as a fee from the escrow account and sent to the treasury. However, while the code transfers this fee to the treasury, it does not properly decrement the corresponding amount from the market's internal accounting of `sy_balance`.

This could lead to discrepancies between the market's internal accounting of `sy_balance` and the actual amount of `SY` balance the market has.

### Remediation

Ensure that whenever an amount is deducted from the escrow in `TradePt`, it is also subtracted from the market's internal `sy_balance`.

### Patch

Fixed in PR#711.

## Escrow Balance Mismanagement HIGH OS-EXF-ADV-02

### Description

In the `WithdrawLp` instruction in `exponent_core`, `do_transfer_lp_out` transfers LP tokens from the market's escrow account to the user's destination account (`token_lp_dst`). However, the `market.lp_escrow_amount`, which represents the total amount of LP (Liquidity Provider) tokens held in escrow by the market, is not decremented after the withdrawal.

If `market.lp_escrow_amount` is not updated (decremented) after the transfer, the protocol will continue to consider the withdrawn tokens as part of the market's liquidity pool. This will lead to improper distribution of emissions from the SY program and farms.

>_ *exponent_core/src/instructions/market_two/withdraw_lp.rs* RUST

```rust
/// Transfer LP tokens from escrow to dst
fn do_transfer_lp_out(&self, amount: u64) -> Result<()> {
    #[allow(deprecated)]
    token_2022::transfer(
        self.transfer_lp_out_context()
            .with_signer(&[&self.market.signer_seeds()]),
        amount,
    )
}
```

### Remediation

Ensure that `market.lp_escrow_amount` is decremented by the amount of LP tokens withdrawn.

### Patch

Fixed in PR#710.

## Utilization of Stale State Values HIGH OS-EXF-ADV-03

### Description

The vulnerability in `claim_farm_emissions` and `market_collect_emission` instructions stems from the failure to update the emission indexes, which results in the retrieval of stale reward values. Both the `claim_farm_emissions` and `market_collect_emission` instructions rely on emission indexes to calculate the rewards for liquidity providers (LPs). Since the emission indexes are not updated, the system still utilizes the stale emission index values, which results in inaccurate reward claims.

>_ *marginfi_standard/src/instructions/read/get_position.rs* RUST

```rust
pub fn to_position_state(&self) -> PositionState {
    PositionState {
        owner: self.position.owner,
        sy_balance: self.position.amount,
        emissions: self
            .position
            .reward_indexes
            .iter()
            .map(|reward| Emission {
                mint: reward.mint,
                amount_claimable: reward.claimable_rewards_amount,
                last_seen_emission_index: reward.last_seen_share_index,
            })
            .collect(),
    }
}
```

Similarly, the `get_position` instruction within `marginfi_standard` (shown above) fails to refresh emissions before returning the `PositionState`. This results in inconsistencies in emission updates and affects subsequent actions such as `deposit_lp` and `withdraw_lp` in `exponent_core`.

>_ *market_two/market_collect_emission.rs* RUST

```rust
pub fn handler(ctx: Context<MarketCollectEmission>, emission_index: u16) -> Result<()> {
    let emission_index = emission_index as usize;
    assert_eq!(
        ctx.accounts.market.emissions.trackers[emission_index].token_escrow,
        ctx.accounts.token_emission_escrow.key()
    );
    let amount = ctx.accounts.lp_position.emissions.trackers[emission_index].staged;
    ctx.accounts.lp_position.emissions.trackers[emission_index].staged = 0;
    [...]
}
```

Also, in the `MarketCollectEmission` instruction, the `last_seen_staged` field in `market.emissions.trackers` is not reduced by the claimed amount. The `last_seen_staged` value keeps track of the total rewards that were last seen in the emissions tracker. By not reducing this value after emission claims, the tracker continues to reflect an incorrect balance, essentially representing more tokens than are actually available.

### Remediation

Ensure that the emission index is updated to reflect the most current market state in the `claim_farm_emissions`, `market_collect_emission`, and `get_position` instructions. Additionally, the `last_seen_staged` field in `market.emissions.trackers` should be reduced by the claimed amount. This ensures that the tracker reflects the actual rewards remaining for future claims and prevents the over-allocation of rewards.

### Patch

The issue concerning the missing index updates was acknowledged.

The issue in `get_position` was resolved in PR#612 and PR#643.

The issue regarding the failure to update the `last_seen_staged` field in `market.emissions.trackers` was resolved in PR#530.

## Locked Escrow Funds HIGH OS-EXF-ADV-04

### Description

The `buy_yt` instruction is missing a critical step: the `SY` (Synthetic Yield) tokens that are borrowed are not properly deposited back into the pool, rendering the funds locked in `token_sy_escrow`. In `buy_yt`, once the PT (Principal Tokens) are sold, the `SY` tokens obtained are supposed to be utilized to repay the borrowed amount. This repayment is done through `do_repay_sy`, which transfers `SY` from the trader's account back to the `token_sy_escrow` account.

This escrow temporarily holds `SY` tokens. However, since the `do_deposit_sy` step is missing, the `SY` tokens remain in the `token_sy_escrow` account. This implies that they are not properly returned to the liquidity pool, where they should be available for future operations.

### Remediation

Add a `do_deposit_sy` `CPI` call after the `do_repay_sy` call. This will properly transfer the repaid `SY` tokens from `token_sy_escrow` back into the `SY` liquidity pool, ensuring that the borrowed funds are fully repaid and available for future utilization in the market.

### Patch

Resolved in PR#558.

## Mathematical Errors from Incorrect Operations HIGH OS-EXF-ADV-05

### Description

In `MarketFinancials::trade_pt`, in the `sy_fee` calculation, `asset_fee` is currently multiplied by the `sy_exchange_rate`. This multiplication will incorrectly scale up the fee, resulting in an incorrect fee calculation. Instead, it should be divided by the exchange rate, which converts the asset fee into `SY` terms, as it effectively scales down the fee according to how many base assets are equivalent to one `SY`. Additionally, in `sy_magnitude_from_net_trader_asset` (shown below), `ceil` should be applied to `sy_magnitude` when `net_trader_asset` is negative, as flooring will round down, potentially resulting in the allocation of fewer `SY` than intended.

>_ *exponent_core/src/state/market_two.rs* RUST

```rust
fn sy_magnitude_from_net_trader_asset(net_trader_asset: DNum, sy_exchange_rate: Number) -> u64 {
    let asset_magnitude: u64 = net_trader_asset.value.floor().abs().try_into().unwrap();
    let sy_magnitude = Number::from_natural_u64(asset_magnitude) / sy_exchange_rate;
    let sy_magnitude = sy_magnitude.floor_u64();

    sy_magnitude
}
```

Additionally, in the `dec_num` library, `MAX_U96` is incorrect. The maximum value for an unsigned integer with `n` bits is $2^n - 1$. For 96 bits, the maximum value is $2^{96} - 1$. Utilizing `1 << 96` gives $2^{96}$, which is one more than this maximum value. This results in an off-by-one error in representing the maximum value for 96-bit unsigned integers.

### Remediation

Ensure that the `sy_fee` is calculated by dividing the `asset_fee` by the `sy_exchange_rate` to convert the fee from the base asset to `SY` units correctly, and that `ceil` is applied to `sy_magnitude` when `net_trader_asset` is negative.

### Patch

1. The issue in `MarketFinancials::trade_pt` was resolved in PR#559.
2. The issue in `sy_magnitude_from_net_trader_asset` was resolved in PR#560.
3. The issue in `dec_num` was resolved in PR#553.

## Improper Reward Distribution MEDIUM OS-EXF-ADV-06

### Description

`LpFarm::increase_share_indexes` calculates how much each emission's share index should be updated based on the passage of time and the total LP (Liquidity Provider) supply. It applies this update to all emissions on the farm. However, it utilizes the total `lp_supply` when adjusting the share indexes, while rewards are meant to be distributed based on specific LP positions rather than the total supply.

>_ *exponent_core/src/state/market_two.rs* RUST

```rust
pub fn update_emissions_from_position_state(
    &mut self,
    position_state: &PositionState,
    lp_supply: u64,
) {
    for (index, current_position) in position_state.emissions.iter().enumerate() {
        let difference =
            current_position.amount_claimable - self.emissions.trackers[index].last_seen_staged;
        let amount_to_increase = Number::from_natural_u64(difference)
            .checked_div(&Number::from_natural_u64(lp_supply))
            .unwrap();
        self.emissions.trackers[index].lp_share_index += amount_to_increase;
        self.emissions.trackers[index].last_seen_staged = current_position.amount_claimable;
    }
}
```

Similarly, `MarketTwo::update_emissions_from_position_state` (shown above) updates the LP share index for emissions based on the `amount_claimable` for each position in `position_state`, using the total `lp_supply` to calculate the increase amount. Thus, both functions assume that rewards should be distributed across the entire LP supply rather than being collected solely for LP tokens deposited in `lp_positions`.

### Remediation

Modify the reward distribution logic to utilize the actual number of LP tokens deposited in active positions rather than the total LP token supply.

### Patch

Resolved in PR#613.

## Inconsistent Handling of Interest Rate Adjustments MEDIUM OS-EXF-ADV-07

### Description

The vulnerability concerns a flaw in the way `vault` handles interest adjustments when the exchange rate fluctuates, specifically in emergency scenarios. When the exchange rate increases, the `SY` amount that was previously tied to `PT` (Principal Tokens) moves to an `uncollected_sy` pool. This is a mechanism to account for the fact that more `SY` is now required for the same amount of `PT`. In the opposite scenario, when the exchange rate decreases (negative interest case), there is protection to prevent interest amounts from being reduced.

While protections exist to prevent a decrease in the `SY` amount when the exchange rate falls, there are no corresponding checks to prevent the `sy_for_pt` amount from increasing.

>_ *vault/stage_yield.rs* RUST

```rust
pub fn handle_stage_yt_yield(
    vault: &mut Vault,
    vault_yield_position: &mut YieldTokenPosition,
    user_yield_position: &mut YieldTokenPosition,
    sy_state: &SyState,
    now: u32,
) -> Result<()> {
    // update vault indexees from SY state
    // and stage any yield to the vault's robot account
    update_vault_yield(vault, vault_yield_position, now, sy_state);

    // TODO - consider removing this check, since deeper in the stack we check for this
    require!(
        !vault.is_in_emergency_mode(),
        ExponentCoreError::VaultInEmergencyMode
    );

    yield_position_earn(vault, user_yield_position, sy_state);

    // Set SY for PT
    vault.set_sy_for_pt();

    Ok(())
}
```

Normally, `update_vault_yield`, which manages these calculations, is not allowed to execute in emergency mode as shown above. Emergency mode is designed to prevent unauthorized or incorrect adjustments during critical conditions. However, in `withdraw_yt` and `merge`, the emergency status of the vault is not verified as in `update_vault_yield`. Consequently, an attacker may exploit this by withdrawing more `sy_amount` than they should be able to, based on the current state of the vault.

### Remediation

Ensure that `withdraw_yt` and `merge` verify the emergency status of the vault before proceeding with operations that affect `SY` amounts.

### Patch

Resolved in PR#536 and PR#548.

## Improper Authorization Login in Farming Instructions MEDIUM OS-EXF-ADV-08

### Description

The `add_farm` and `send_farm_tokens` instructions in `exponent_core::market_two` lack sufficient access control checks. Without proper access controls, any user may call these functions, potentially manipulating the farm emissions data, including adding or modifying farms in the emissions array. Without restricted access, any user may repeatedly call the `add_farm` instruction to add new entries to the `farm_emissions` array, bloating the array and unnecessarily consuming storage resources.

>_ *market_two/admin/send_farm_tokens.rs* RUST

```rust
#[derive(Accounts)]
pub struct SendFarmTokens<'info> {
   [...]
    #[account(
        mut,
        associated_token::mint = mint,
        associated_token::authority = market.token_sy_escrow,
    )]
    pub token_farm: InterfaceAccount<'info, TokenAccount>,
    pub token_program: Interface<'info, TokenInterface>,
}
```

Furthermore, in the `SendFarmTokens` structure in `send_farm_tokens`, the authority of the `token_farm` account is incorrectly set to `market.token_sy_escrow` instead of `market` itself. This would lead to following instruction calls getting reverted.

### Remediation

Enforce proper access control to ensure that only authorized users are able to invoke `add_farm` and `send_farm_tokens`. Additionally, set the authority to `market` for the `token_farm` account.

### Patch

Resolved in PR#526 and PR#527.

## Faulty Reallocation of Market Size MEDIUM OS-EXF-ADV-09

### Description

The vulnerability concerns the incorrect handling of CPI (Cross-Program Invocation) accounts during the reallocation of the `MarketTwo` account in the `AddMarketEmission` instruction. Specifically, the issue is that the `market.cpi_accounts` are utilized for reallocation instead of the `cpi_accounts` provided as input to the handler function.

>_ *market_two/admin/add_market_emission.rs* RUST

```rust
pub fn update_market(&mut self, cpi_accounts: CpiAccounts) {
    self.market.cpi_accounts = cpi_accounts;
}
```

This implies that when the account is resized, it relies on the existing `cpi_accounts` in the `market` account. Ideally, the `realloc` should be based on the `cpi_accounts` passed in through the handler function input, not on the current state of the `market` account. If the `market.cpi_accounts` are outdated or incorrect, reallocating the account based on this data may result in an incorrect size calculation. Thus, there will be insufficient space for the new `MarketEmission` data.

### Remediation

Ensure that the `realloc` operation accurately reflects the size requirements based on the input `cpi_accounts` provided in the handler function.

### Patch

Resolved in PR#529.

## Missing Share Index Update MEDIUM OS-EXF-ADV-10

### Description

In the `market_two::modify_market_setting` instruction, the `farm_emission` rates are updated without properly adjusting the relevant share indexes. Share indexes track how rewards are distributed to participants. These indexes ensure that when the emission rate changes, users still receive the correct amount of rewards based on their share of the total pool and the time they have participated. Here, the `ChangeFarmRate` action is updating the emission rate ( `new_rate` ) for a farm, but it is not properly adjusting the share indexes beforehand, which may result in incorrect reward calculations.

Failing to update the share indexes before modifying the farm emission rate can lead to unfair and incorrect reward distributions. Participants who were in the farm before the rate change may not receive their full rewards. Since the share indexes were not updated, rewards earned before the rate change are lost or miscalculated, resulting in incorrect reward amounts for those users. This inconsistency arises because the system has two different emission rates without properly accounting for the transition between them.

### Remediation

Ensure the program calls `increase_share_indexes` to update the share indexes based on the current emission rate and timestamp.

### Patch

Resolved in PR#528.

## Unutilized Accumulated Funds MEDIUM OS-EXF-ADV-11

### Description

`increase_lambo_fund` in the vault updates the `treasury_sy` and `treasury_emission` fields within the vault account. However, in the current implementation, `treasury_sy` and `treasury_emission` are never retrieved, accessed, or utilized in any subsequent logic after they are updated in `increase_lambo_fund`. Consequently, the surplus emission in `earned_emission_surpluses` and the surplus amount of `SY` are never actually utilized in the treasury, resulting in unnecessary wastage of funds.

>_ *exponent_core/src/state/vault.rs* RUST

```rust
pub fn increase_lambo_fund(&mut self, earn_all_result: &EarnAllResult) {
    self.treasury_sy = self
        .treasury_sy
        .checked_add(earn_all_result.earned_sy_surplus)
        .unwrap();
    for (index, earned_emission) in earn_all_result.earned_emission_surpluses.iter().enumerate()
    {
        self.emissions[index].treasury_emission = self.emissions[index]
            .treasury_emission
            .checked_add(*earned_emission)
            .unwrap();
    }
}
```

### Remediation

Ensure that the vault logic is updated so that the accumulated amounts in `treasury_sy` and `treasury_emission` are correctly utilized.

### Patch

Resolved in PR#521.

## Incorrect Flooring Conversion MEDIUM OS-EXF-ADV-12

### Description

`floor_u128` is supposed to convert a high-precision number (represented as a `Number` in the form of a `PreciseNumber`, which utilizes `U256` for its internal representation) into a `u128` by rounding down (flooring) to the nearest integer. However, in this case, the function fails to correctly apply the floor operation.

>_ *libraries/precise_number/src/lib.rs* RUST

```rust
pub fn floor_u128(&self) -> u128 {
    self.to_pn().to_imprecise().unwrap()
}
```

In the test case given below, `Number::from_ratio(19, 10)` creates a `Number` equivalent to 1.9. The expected behavior of `floor_u128` is to return 1 (the largest integer not greater than 1.9). However, since `to_imprecise` does not correctly floor the number but instead performs a direct truncation without regard to the fractional part, the test fails.

>_ *test.rs* RUST

```rust
#[test]
fn test_asdf() {
    assert_eq!(Number::from_ratio(19, 10).floor_u128(), 1);
}
```

### Remediation

Ensure that the number is explicitly floored before converting it to a `u128`.

### Patch

Resolved in PR#519.

# 05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

| ID | Description |
|---|---|
| OS-EXF-SUG-00 | `trade` does not verify whether the exchange rate, after accounting for fees, remains greater than one. This may allow unfavorable trades where the user ends up receiving less value than the amount of `PT` traded. |
| OS-EXF-SUG-01 | There are several instances where proper validation is not done, resulting in potential security issues. |
| OS-EXF-SUG-02 | `MintSy` allows any pre-existing token account to be accepted as `token_base_account_authority`, which is inefficient and increases complexity. |
| OS-EXF-SUG-03 | `AddPrincipleAdmin` reallocates 64 bytes for adding an admin, which is more than the 32 bytes needed for a public key, potentially wasting space. |
| OS-EXF-SUG-04 | Recommendation for modifying the codebase for improved functionality, efficiency, and maintainability. |
| OS-EXF-SUG-05 | Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices. |
| OS-EXF-SUG-06 | The codebase contains multiple instances of unutilized and redundant code that should be removed for better maintainability and clarity. |
| OS-EXF-SUG-07 | The codebase contains multiple cases of unnecessary code that should be removed for better maintainability and clarity. |
| OS-EXF-SUG-08 | The overall code may be streamlined further to reduce complexity, eliminate redundancy, and enhance readability. |

## Faulty Post-Fee Rate Validation OS-EXF-SUG-00

### Description

`trade` currently checks that the exchange rate is greater than one with the assertion `assert!(er > N::one()`. The calculation of the `net_trader_asset` considers the impact of the transaction fee. If the fee is not appropriately considered, the post-fee exchange rate may be less than one, resulting in an incorrect valuation. `trade` computes `pre_fee_net_trader_asset` utilizing the exchange rate but does not validate if the resulting value (after applying the fee) maintains the expected valuation relationship.

>_ *solana/libraries/tcurve/src/math.rs* RUST

```rust
pub fn trade<N: Num>(
   [...]
) -> TradeResult<N> {
    [...]
    let er = exchange_rate(l_p, rate_scalar, rate_anchor);
    assert!(er > N::one(), "Asset cannot be worth less than PT");
    // negate the trader PT to get the net change in asset for the trader
    let pre_fee_net_trader_asset = -net_trader_pt / er;
    let fee = asset_fee(pre_fee_net_trader_asset, fee_rate);

    // subtract the fee from the net trader asset
    // if net_trader_asset is negative, the user is buying PT and selling asset and the "fee"
        ↪  value is positive in order to increase the magnitude of net_trader_asset (increasing
        ↪  the amount of asset the user must pay)
    // if net_trader_asset is positive, the user is selling PT and buying asset and the "fee"
        ↪  value is positive in order to decrease the magnitude of net_trader_asset
    let net_trader_asset = pre_fee_net_trader_asset - fee;
    [...]
}
```

The function does not revalidate whether the post-fee exchange rate, which determines the final net asset value after accounting for fees, still satisfies the condition that the asset value is greater than `PT`. This lack of revalidation can lead to scenarios where traders receive less value than expected, especially if the fee is high or if there are discrepancies between the actual and expected exchange rates after the fee deduction.

### Remediation

Move the `er > 1` check to `exchange_rate` to ensure that the exchange rate calculation itself enforces the condition that the asset is worth more than `PT`.

### Patch

The issue was acknowledged.

## Missing Validation Logic OS-EXF-SUG-01

### Description

1. Currently, mint checks are not properly enforced for token accounts. Instead, the system relies on these accounts to fail during token transfers if the mint is incorrect. This is a reactive approach that may result in runtime errors. Additionally, verify `mint_sy` in the `initialize_vault` instruction, and the mint and owner of the `token_emission_escrow` associated token account in `MarketCollectEmission`.
2. In `MarketTwo::add_farm`, there is no check to see if the `token_mint` already exists in the market's list of farms. This may result in the addition of duplicate farms for the same `token_mint`, especially because `get_farm_emission_index` does not properly handle duplicates.
3. It is advisable to verify the `vault` account against `market.vault` in both `BuyYt` and `SellYt` instructions to ensure that the correct vault is utilized.
4. Add a check in the `exponent_core::AddEmission` instruction that verifies the mint of the `robot_token_account`, as it may be modified later, which would result in a Denial of Service (DoS) attack if the wrong input is provided. This check will ensure that the token account provided by the user matches the expected token type.

### Remediation

1. Explicitly check the mints for all token accounts.
2. Verify whether a farm with the same `token_mint` already exists.
3. Implement the above check.
4. Add the missing validations mentioned above.

### Patch

1. The Issue in `MarketCollectEmission` was resolved in PR#550 and the other two issues were acknowledged.
2. Issue #2 was resolved in PR#552.
3. Issue #3 was resolved in PR#551.
4. Issue #4 was resolved in PR#514.

## Code Optimization OS-EXF-SUG-02

### Description

1. The `has_one` constraint ensures that the specified field of an account matches the public key of a given account, verifying that the account is correctly linked to the expected account. This check may be utilized in `token_sy_escrow` and `token_sy_treasury` accounts in the `CollectInterest` instruction to simplify the constraints.

   >_ *exponent_core/src/instructions/vault/collect_interest.rs* RUST

   ```rust
   pub struct CollectInterest<'info> {
       [...]
       #[account(
           mut,
           address = vault.escrow_sy,
       )]
       pub token_sy_escrow: InterfaceAccount<'info, TokenAccount>,

       #[account(
           mut,
           address = vault.treasury_sy_token_account,
       )]
       pub token_sy_treasury: InterfaceAccount<'info, TokenAccount>,
       [...]
   }
   ```

2. In its current implementation, the `add_emission` and `add_market_emission` instructions update the entire `CpiAccounts` each time, which is an unoptimized method and increases the risk of potentially unexpected errors.

### Remediation

1. Incorporate the `has_one` check in the `token_sy_escrow` and `token_sy_treasury` accounts in the `CollectInterest` instruction.
2. Push a new `CpiInterfaceContext` instead of updating the entire `CpiAccounts` each time for better efficiency and reduced risk of errors.

### Patch

1. Issue #1 resolved in PR#523.
2. Issue #2 has been acknowledged.

## Efficient Memory Reallocation OS-EXF-SUG-03

### Description

In the context of `AddPrincipleAdmin`, each time a new admin is added, the account is reallocated with an additional 64 bytes. However, a single public key is only 32 bytes in size. This implies that half of the reallocated space (32 bytes) is unutilized for every addition. Over multiple additions, this results in a significant waste of memory.

>_ *exponent_admin/src/lib.rs* RUST

```rust
#[derive(Accounts)]
pub struct AddPrincipleAdmin<'info> {
    #[account(
        mut,
        realloc = admin_account.try_to_vec().unwrap().len() + 64,
        realloc::payer = fee_payer,
        realloc::zero = false
    )]
    pub admin_account: Account<'info, Admin>,
    /// CHECK:
    pub new_admin: UncheckedAccount<'info>,
    #[account(mut)]
    pub fee_payer: Signer<'info>,
    pub uber_admin: Signer<'info>,
    pub system_program: Program<'info, System>,
}
```

Additionally, when an admin is removed via `RemovePrincipleAdmin`, the public key is removed from the list of administrators. This results in unutilized space in the account because the allocated memory does not shrink automatically. The account still occupies the same amount of space, even though some of it is no longer utilized.

### Remediation

Calculate the difference between the current lamports and the lamports required for the new size after reallocating, and adjust the account's lamports accordingly. This process will be more efficient than reallocating each time a public key is added or removed.

### Patch

Resolved in PR#511.

## Code Refactoring OS-EXF-SUG-04

### Description

1. In the current implementation, it is impossible to update `vault.cpi_accounts` without adding new emissions in the `ModifyVaultSetting` instruction, which is not a feasible option. The `cpi_accounts` field in the `Vault` structure is crucial and may result in a denial-of-service attack in the case of incorrect inputs during initialization.
2. Store all Program Derived Address (PDA) seed strings as constants rather than directly utilizing them to improve code maintainability and reduce the risk of errors during future edits.
3. Within `exponent_core`, rename the owner field to `depositor` in the `DepositYtEvent` structure, as the depositor may not always be the actual owner of the `user_yield_position`.
4. Round up the `sy_in` value in the `if` block and the `pt_in` value in the `else` block in `tcurve::add_liquidity` so as to slightly favor the protocol by ensuring that liquidity providers (LPs) contribute a bit more than the minimum required.
5. The `new_lp_supply` field in the `DepositLiquidityEvent` may not immediately reflect the correct updated supply of LP tokens after minting new tokens. This is because the LP token mint account ( `mint_lp` ) may not have been updated with the newly minted amount by the time the event is emitted.
6. The `address_lookup_table` parameter in the `MarketTwoInit` instruction may be redundant compared to the `address_lookup_table` passed in the context (the account defined in the `MarketTwoInit` structure). If these two values are not aligned, it will result in inconsistencies during transaction execution.

### Remediation

1. Update the `ModifyVaultSetting` instruction by adding an option that allows admins to update `vault.cpi_accounts` directly.
2. Utilize constants to store the PDA seeds.
3. Rename the owner field to `depositor`.
4. Round up to favor the protocol.
5. Add the newly minted LP tokens to the existing supply in the event log, ensuring `new_lp_supply` accurately reflects the total LP token supply after the minting process.
6. Remove the redundant `address_lookup_table` parameter in the `MarketTwoInit` instruction.

**Patch**

1. Issue #1 resolved in PR#617.
2. Issue #2 resolved in PR#616.
3. Issue #3 resolved in PR#618.
4. Issue #4 resolved in PR#862.
5. Issue #5 resolved in PR#863.
6. Issue #6 resolved in PR#863.

## Code Maturity OS-EXF-SUG-05

### Description

1. Whitelist the `sy_program` in the `initialize_vault` instruction with the public keys of `marginfi-standard` and `kamino-lend-standard` as a security enhancement to restrict the programs that may interact with the vault.
2. To ensure proper reallocation in the `WithdrawYt` instruction, utilize `realloc` so that `WithdrawYt` is equipped to handle dynamic changes in the size of `user_yield_position`.
3. Utilize dedicated functions such as `inc_pt_balance`, `dec_pt_balance`, `inc_sy_balance`, and `dec_sy_balance` in order to enhance code clarity and maintainability in `MarketFinancials::trade_pt`.
4. In `YieldTokenPosition::earn_emissions`, it would be appropriate to set `e.last_seen_index` to `emission.final_index` instead of `emission.last_seen_index` for better accuracy.

   >_ *exponent_core/src/state/yield_token_position.rs* RUST

   ```rust
   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() {
           let e = &mut self.emissions[index];
           let earned_emission =
               calc_share_value(e.last_seen_index, emission.final_index, sy_balance);
           e.inc_staged(earned_emission);
           e.last_seen_index = emission.last_seen_index;
       }
   }
   ```

5. The comment on the `authority_klend_account` field in the `SyMeta` structure incorrectly states *"Authority over the Marginfi account"* instead of *"Authority over the Kamino account"*. Update the comment to reflect the correct account.

### Remediation

Implement the above-mentioned suggestions.

### Patch

1. Issue #1 has been acknowledged.
2. Issue #2 resolved in PR#522.
3. Issue #3 resolved in PR#554.
4. Issue #4 resolved in PR#620.
5. Issue #5 resolved in PR#517.

## Code Redundancy OS-EXF-SUG-06

### Description

1. `collect_emission_from_position_state` and `approx_pt_for_exact_sy` in `market_two` are not utilized and may be removed.
2. In the `MarketEmission::SIZE` calculation, the size allocated for `last_seen_global_index` is unnecessary.
3. Within `deposit_yt::handle_deposit_yt` in `exponent_core`, it is unnecessary to check if the vault is active for calling `vault.set_sy_for_pt` since this is already checked in `validate`.

>_ *exponent_core/src/instructions/vault/deposit_yt.rs* RUST

```rust
pub fn handle_deposit_yt(
    vault: &mut Vault,
    vault_yield_position: &mut YieldTokenPosition,
    user_yield_position: &mut YieldTokenPosition,
    sy_state: &SyState,
    now: u32,
    amount: u64,
) -> Result<()> {
    [...]
    if vault.is_active(now) {
        vault.set_sy_for_pt();
    }
    Ok(())
}
```

### Remediation

Remove the redundant and unutilized code instances highlighted above.

### Patch

1. Issue #1 resolved in PR#555.
2. Issue #2 resolved in PR#556.
3. Issue #3 resolved in PR#619.

## Unutilized Code OS-EXF-SUG-07

### Description

1. The `fee_payer` account in the `SendFarmTokens` instruction appears unused and may be removed.
2. The `remaining_staged` field in the `MarketCollectEmissionEvent` structure is always set to zero since all staged emissions are cleared. Thus, it does not serve any purpose and should be removed.

### Remediation

Remove the above instances of unutilized code.

### Patch

1. Issue #1 resolved in PR#864.
2. Issue #2 resolved in PR#866.

## Code Clarity

OS-EXF-SUG-08

### Description

1. The `DepositLiquidity` instruction structure is misspelled as `DepositLiquidity`. Ensure to modify the name to reflect the correct spelling.
2. The `owner` field in `DepositLpEvent` may not be an appropriate name, as the `owner` may not actually be the owner of the `LpPosition` passed to the instruction. Consider a more suitable variable name.
3. Update the comment in the `TradePt` instruction (*"net_trader_sy and net_trader_pt must have the same sign"*), as currently, it is incorrect.

### Remediation

Implement the above-mentioned changes.

### Patch

1. Issue #1 resolved in PR#867.
2. Issue #2 was acknowledged.
3. Issue #3 resolved in PR#830.

# A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

**CRITICAL**

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

**HIGH**

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

**MEDIUM**

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

**LOW**

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

**INFO**

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

# B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.