Table of contents

Contest Summary

Sponsor: First Flight #41

Dates: May 29th, 2025 - Jun 5th, 2025

[See more contest details here]()

Results Summary

Number of findings:

- High: 5
- Medium: 3
- Low: 0

High Risk Findings

H-01. Missing Decimal Handling in LP Token Mint Calculation

Description

The `liquidity_calculation` function calculates the LP tokens to mint as the square root of the product of the raw token amounts. However, it does not account for differences in token decimals, which leads to incorrect LP token amounts when tokens have different decimal precisions.

Impact

- Incorrect LP token minting causing unfair liquidity representation.
- Potential economic exploits due to miscalculated LP shares.
- Pool imbalance and user losses.

Recommendation

Normalize token amounts by their decimals before performing the liquidity calculation to ensure accurate LP token minting.

Fixed Code

```
fn liquidity_calculation(

amount_token_a: u64,

decimals_token_a: u8,

amount_token_b: u64,

decimals_token_b: u8,

lp_token_decimals: u8,

) -> Result<u64> {

// Convert amounts to u128 for safe math

let amount_a_u128 = amount_token_a as u128;

let amount_b_u128 = amount_token_b as u128;



// Normalize token amounts to a common decimal scale (e.g., 18 decimals)

// Adjust each amount by shifting decimals to 18

let scale = 18u32;

let adjusted_amount_a = amount_a_u128

.checked_mul(10u128.pow((scale - decimals_token_a as u32) as u32))

.ok_or(AmmError::Overflow)?;

let adjusted_amount_b = amount_b_u128

```

```
.checked_mul(10u128.pow((scale - decimals_token_b as u32) as u32))

.ok_or(AmmError::Overflow)?;



// Calculate lp amount with normalized amounts

let lp_amount_u128 = adjusted_amount_a

.checked_mul(adjusted_amount_b)

.ok_or(AmmError::Overflow)?

.integer_sqrt();



// Adjust lp amount back to lp_token_decimals

let lp_amount_scaled = lp_amount_u128

.checked_div(10u128.pow((scale - lp_token_decimals as u32) as u32))

.ok_or(AmmError::Overflow)?;



if lp_amount_scaled == 0 {

return err!(AmmError::LpAmountCalculation);

}



Ok(lp_amount_scaled as u64)

}
```

This function takes the decimals of each token and the LP token as input, normalizes the token amounts to a common precision, performs the square root calculation, and scales the LP token amount accordingly.

H-02. Incorrect LP Token Calculation in `provide_liquidity` Instruction

Description

The `provide_liquidity` instruction uses the same `liquidity_calculation` function as the `initialize_pool` instruction to calculate the amount of LP tokens to mint based on the amounts of tokens A and B provided:

```

let lp_to_mint: u64 = liquidity_calculation(amount_a, amount_b)?;

```

This calculation is only correct during the **initial pool creation** (when no LP tokens exist yet). For subsequent liquidity additions, the LP token amount must be proportional to the existing LP supply and reserves.

Impact

Using the initial pool creation formula to mint LP tokens on additional liquidity deposits will cause:

- **Incorrect LP token minting:** The amount of LP tokens minted will not reflect the depositor holdings.
- **Pool state inconsistency:** Pool accounting and invariant assumptions may break, causing downstream issues in swaps or removals.

Correct LP Token Calculation for Adding Liquidity

When liquidity already exists, LP tokens to mint must be proportional to the existing LP supply and token reserves. The correct formula is:

```

lpA = (amountA * totalSupply) / reserveA;

lpB = (amountB * totalSupply) / reserveB;

lpTokensToMint = min(lpA, lpB);

```

Where:

- `totalSupply` is the current total supply of LP tokens.
- `reserveA` and `reserveB` are the current token balances in the pool vaults.

Recommended Fix

Replace the incorrect call to `liquidity_calculation` with a function that:

1. Fetches the current total LP token supply.
2. Fetches the current vault balances (reserves) of tokens A and B.
3. Calculates `lp_to_mint` as the minimum of the proportional LP tokens derived from each token amount relative to reserves and total LP supply.

Fix code:

```

fn calculate_lp_tokens_to_mint(

amount_a: u64,

amount_b: u64,

reserve_a: u64,

reserve_b: u64,

total_lp_supply: u64,

) -> Result<u64> {

let lp_a = (amount_a as u128)

.checked_mul(total_lp_supply as u128)

.ok_or(AmmError::Overflow)?

.checked_div(reserve_a as u128)

.ok_or(AmmError::DivideByZero)?;



let lp_b = (amount_b as u128)

.checked_mul(total_lp_supply as u128)

.ok_or(AmmError::Overflow)?

.checked_div(reserve_b as u128)

.ok_or(AmmError::DivideByZero)?;


```

```rust

let lp_to_mint = lp_a.min(lp_b);



if lp_to_mint == 0 {

return err!(AmmError::LpAmountCalculation);

}



Ok(lp_to_mint as u64)

}

```

H-03. Missing Slippage Protection in `provide_liquidity` instruction

Description

The `provide_liquidity` function fails to implement slippage protection when calculating the required amount of token B for a given `amount_a` of token A. Specifically:

1. Users supply `amount_a` without specifying a maximum acceptable amount for token B (`max_amount_b`)
2. The calculated `amount_b` (via `calculate_token_b_provision_with_a_given`) is used unconditionally
3. No validation occurs to ensure the exchange rate is within user expectations

This allows scenarios where:

- A malicious actor could sandwich the transaction (front/back-run) to manipulate prices
- Users receive an unfavorable exchange rate due to high slippage
- Token B provision could become economically nonviable (e.g., 1 unit of token B for 1,000,000 token A)

Code Proof

```
pub fn provide_liquidity(context: Context<ModifyLiquidity>, amount_a: u64) -> Result<()> {

let amount_b = calculate_token_b_provision_with_a_given(

&mut context.accounts.vault_a,

&mut context.accounts.vault_b,

amount_a

)?; // No validation of amount_b



// (Token transfer logic would follow here)

}

```

Impact

- **Financial Loss:** Users may receive significantly less token B than expected
- **MEV Exploitation:** Traders could extract value through price manipulation
- **System Abuse:** Attackers could intentionally create unfavorable pools to drain users
- **Trust Degradation:** Users lose confidence in the protocol's safety mechanisms

Recommendation

Implement slippage protection with a user-defined maximum:

```
pub fn provide_liquidity(
context: Context<ModifyLiquidity>,
amount_a: u64,
+ max_amount_b: u64 // Add slippage tolerance parameter
) -> Result<()> {
let amount_b = calculate_token_b_provision_with_a_given(...)?;
+ require!(
+ amount_b <= max_amount_b,
+ LiquidityError::ExcessiveTokenBRequest
+ );


// Proceed with transfers
}
```

H-04. Missing Account Loading in `provide_liquidity` instruction

Description

The `provide_liquidity` instruction calls accounts and performs operations on account structs (e.g., vaults) without explicitly loading them, which is unusual in Anchor if using AccountLoader or if state mutation is expected.

Impact

May cause logic inconsistencies if mutation or checks rely on up-to-date account data.

Could be an indicator of incorrect Anchor account modeling, especially if state-modifying methods or field access are assumed.

Recommendation

Ensure all accounts that require access to their internal state or mutation are loaded properly using:

```

context.accounts.liquidity_provider_lp_account.reload()?;

context.accounts.lp_mint.reload()?;

context.accounts.vault_a.reload()?;

context.accounts.vault_b.reload()?;

```

H-05. Incorrect formula and fee calculation in `swap_exact_in` instruction

Description

The `swap_exact_in` instruction calculates the output token amount and fees incorrectly. Specifically, it does not properly account for token decimals, which can lead to inaccurate swap rates when tokens have different decimal places. Additionally, the fee calculation is applied to the output amount (`amount_out`) rather than the input amount (`amount_in`), which is inconsistent with common AMM designs where fees are deducted from the input before calculating the output. These issues can cause incorrect token amounts to be swapped, resulting in unfair trades and potentially unexpected losses for users.

Infected Code Snippet

```
if zero_for_one {

let numerator: u128 = (reserve_b as u128)

.checked_mul(amount_in as u128)

.ok_or(AmmError::Overflow)?;

let denominator: u128 = (reserve_a as u128)

.checked_add(amount_in as u128)

.ok_or(AmmError::Overflow)?;



if denominator == 0 {

return err!(AmmError::DivisionByZero);

}



let mut amount_out: u64 = numerator.div_floor(&denominator) as u64;



let lp_fees = (amount_out as u128 * 3).div_floor(&1000) as u64;



amount_out = amount_out - lp_fees;

}

```

Impact

- Incorrect handling of decimals may cause the swap output to be skewed, especially for tokens with differing decimals.
- Calculating fees on the output rather than input can distort the actual value exchanged, potentially disadvantaging liquidity providers or traders.
- The combination of these issues can result in loss of funds or unfair exchange rates, degrading user trust and platform reliability.

Recommendation / Fix

1. **Handle token decimals properly:** Normalize token amounts to a common base (e.g., convert to u128 with decimals factored in) before calculations.

2. **Calculate fees on the input amount:** Deduct the fee from `amount_in` before computing the output, as is standard in AMM designs.

3. **Use the correct constant product formula with fees:**

```rust

let fee_numerator = 997u128; // 0.3% fee

let fee_denominator = 1000u128;



if zero_for_one {

let amount_in_with_fee = (amount_in as u128)

.checked_mul(fee_numerator)

.ok_or(AmmError::Overflow)?;



let numerator = amount_in_with_fee

.checked_mul(reserve_b as u128)

.ok_or(AmmError::Overflow)?;



let denominator = (reserve_a as u128)

.checked_mul(fee_denominator)

.ok_or(AmmError::Overflow)?

.checked_add(amount_in_with_fee)

.ok_or(AmmError::Overflow)?;



let amount_out =
numerator.checked_div(denominator).ok_or(AmmError::DivisionByZero)?;

```

```
// amount_out is the final amount after fees

}
```

This approach correctly applies the 0.3% fee on the input amount and then calculates the output amount using the constant product formula. Token decimals should be normalized before these calculations if tokens have different decimal places.

Medium Risk Findings

M-01. Initialization Order Issue Causing `lp_mint` Account Creation to Fail

Description

In the `InitializePool` instruction, the `lp_mint` account is being initialized with seeds that include the `liquidity_pool` key. However, the `liquidity_pool` account itself is also being initialized in the same instruction **after** `lp_mint`. Since `liquidity_pool` is not yet created and assigned a key at the time `lp_mint` is initialized, using its key as a seed causes the `lp_mint` account initialization to fail.

Additionally, Anchor validates and reads accounts sequentially in the order they are declared in the accounts struct s sequential account initialization and fixes the seed derivation issue.

M-02. Vault Account Front-Running Leading to Denial of Service

Description

The vault token accounts `vault_a` and `vault_b` are initialized using the `init` attribute with associated token account seeds derived only from the token mint and liquidity pool authority. These seeds are predictable and not unique per pool initialization, allowing an attacker to pre-create (front-run) the vault accounts before the legitimate pool initialization.

Infected Code

```
#[account(

init,

payer = creator,

associated_token::mint = token_mint_a,

associated_token::authority = liquidity_pool,

associated_token::token_program = token_program

)]

pub vault_a: InterfaceAccount<'info, TokenAccount>,



#[account(

init,

payer = creator,

associated_token::mint = token_mint_b,

associated_token::authority = liquidity_pool,

associated_token::token_program = token_program

)]

pub vault_b: InterfaceAccount<'info, TokenAccount>,
```

Impact

Attackers can pre-initialize vault accounts, causing the pool initialization to fail and resulting in a denial of service by blocking new pool creation.

Recommendation and Fix

Use Program Derived Addresses (PDAs) with additional unique seeds such as the pool address or a nonce to generate vault accounts. This ensures vault account addresses are unique and cannot be pre-created by others.

Fixed Code Snippet

```

#[account(

init,

payer = creator,

seeds = [b"vault_a", liquidity_pool.key().as_ref()],

bump,

token::mint = token_mint_a,

token::authority = liquidity_pool,

token::token_program = token_program,

)]

pub vault_a: Account<'info, TokenAccount>,



#[account(

init,

payer = creator,

seeds = [b"vault_b", liquidity_pool.key().as_ref()],

bump,

token::mint = token_mint_b,

token::authority = liquidity_pool,

token::token_program = token_program,
```

```
)]

pub vault_b: Account<'info, TokenAccount>,

```

M-03. Unsafe Casting from `u128` to `u64` Without Overflow Checks

Description

In the liquidity_operations.rs file, values are cast from `u128` to `u64` without checking whether the value fits within the `u64` range. This can lead to silent truncation and incorrect calculations, especially when large input values are used.

Infected Code

In `liquidity_calculation`:

```

let lp_amount_to_mint: u64 = lp_amount_to_mint_u128 as u64;

```

In `calculate_token_b_provision_with_a_given`:

```

let amount_b_to_deposit: u64 = amount_b_to_deposit_u128 as u64;

```

In `remove_liquidity` instruction:

```

let amount_b_to_return = amount_b_to_return_u128 as u64;

let amount_a_to_return: u64 = amount_a_to_return_u128 as u64;

```

Impact

- A malicious or misconfigured input could trigger an overflow scenario that causes token misallocation, accounting errors, or unexpected program behavior.

- In the worst case, truncated values could result in incorrect minting or burning of tokens, potentially leading to financial loss or denial of service.

Recommendation

Use `.try_into()` or `.try_from()` with proper error propagation instead of direct `as` casting. This ensures the operation fails safely if the value exceeds `u64::MAX`.

Fixed Code Snippet

Example fix for `liquidity_calculation`:

```

let lp_amount_to_mint: u64 = lp_amount_to_mint_u128

.try_into()

.map_err(|_| AmmError::Overflow)?;

```

Apply similar error-handled casting in all other instances of `u128 -> u64` conversions.