Protocol Summary

This protocol is a minimalist Automated Market Maker (AMM) written in Rust for the Solana blockchain using the Anchor framework. It enables users to create liquidity pools between two SPL tokens, perform exact-in and exact-out swaps, add and remove liquidity, and collect trading fees as liquidity providers.

The AMM follows a constant product invariant (x * y = k) and handles all operations through CPI calls to the SPL Token program. It is designed to be educational and modular, serving as a template for forks or extensions with custom logic.

There is no oracle integration, governance, or fee splitting beyond basic LP compensation. Instead, the focus is on correct handling of token balances, slippage protection, and state updates using Solana-native patterns like .reload().

Key Flows and Roles

👤 User

Creates a pool by depositing two SPL tokens.

Adds or removes liquidity and receives/burns LP tokens.

Swaps tokens via exact-in or exact-out operations.

Collects fees accumulated from trading activity.

🧪 Liquidity Operations

Handles:

Proportional liquidity provisioning and withdrawal.

LP token minting and burning logic.

Fee calculation and distribution based on trading volume.

Initial mint protection to prevent inflation when total LP supply is zero.

🔁 Swap Operations

Handles:

Execution of token swaps (exact-in and exact-out).

Slippage tolerance verification (input and output limits).

Balance reloading (.reload()) before calculations to prevent stale reads.

Invariant enforcement (x * y >= k) after swaps.

🏗 Transfer & Account Management

Verifies token accounts and authorities.

Executes CPI calls to the SPL Token program for minting, burning, transferring.

Ensures proper validation of signer permissions and account constraints.

Invariants

The product of reserves (x*y) must not decrease after a swap.

Liquidity additions must be proportional to current pool state.

Only the pool creator can initialize a new pair.

.reload() must be used before any read-modify-write operation on token accounts.

Fee minting must never occur on first liquidity deposit (avoiding LP inflation).

Slippage bounds must be respected during swaps.

Disclaimer

Rubén Cruz has performed a time-boxed audit exercise on this Rust-based AMM as part of a formative security review. The audit focused on core logic, invariant preservation, and interaction with the SPL Token program. No warranties are provided regarding the complete security of the codebase. This report does not represent an endorsement of the protocol's financial or operational model.

Risk Classification

Impact \ Likelihood

High

Medium

Low

Classification based on the CodeHawks severity matrix.

Audit Details

Scope:

src/lib.rs

src/liquidity_operations.rs

src/swap_operations.rs

src/liquidity_pool.rs

src/transfer.rs

Issues Found

Severity of Issues High: 🚨 3 Medium: ⚠ 0 Low: ℹ 0 Total: 3

Stale Token Account State Causes Incorrect Liquidity Calculations

Summary

The add_liquidity and remove_liquidity functions in liquidity_operations.rs perform multiple CPIs to the SPL-Token program—such as transfer_tokens, mint_to, burn, and transfer_checked—that mutate token accounts (vault_a, vault_b) or token mint accounts (lp_mint). However, the contract fails to invoke .reload() on these accounts immediately after the CPIs. This results in the in-memory state of these accounts remaining stale, leading to incorrect reserve or supply values being used in subsequent calculations.

This oversight can cause miscalculated LP issuance or redemption, reserve imbalances, and in extreme cases, arithmetic errors such as overflows or division by zero that disrupt the AMM logic or block transactions.

Vulnerability Details

Anchor frameworks cache the initial deserialized state of all accounts in ctx.accounts. When a CPI mutates the underlying token account or mint on-chain, the cached in-memory structure does not reflect this update unless .reload() is explicitly called. Failing to reload mutated accounts means subsequent logic will operate on outdated data.

For example, in add_liquidity, after a CPI call deposits tokens into vault_a, the value of ctx.accounts.vault_a.amount remains unchanged unless reloaded. If the LP minting logic relies on this stale value to calculate the number of LP tokens to mint, the calculation will be incorrect. Similar issues arise in remove_liquidity if the LP token supply or vault balances are used post-modification without reloading.

This can lead to LP over-issuance, under-redemption, or logic errors when assumptions about account state diverge from actual on-chain state.

Impact

Incorrect reserve or supply values due to stale memory representations can cause:

Over-issuance or under-issuance of LP tokens, diluting other LPs or benefiting attackers.

Reserve imbalance, distorting pool ratios and enabling arbitrage.

Division-by-zero or overflow errors, leading to transaction reverts and potential denial of service.

Loss of user trust, as the protocol behaves unpredictably due to desynchronized internal state.

Proof of Concept

Initial state:

vault_a.amount = 1000

vault_b.amount = 1000

lp_mint.supply = 1000

User deposits amount_a = 100, triggering a transfer CPI to vault_a. On-chain vault_a.amount = 1100, but in-memory it remains 1000. LP tokens are minted based on vault_a.amount = 1000, leading to a miscalculation:

Expected mint amount: (100 * 1000) / 1100 = 90.9...

Actual mint amount using stale data: (100 * 1000) / 1000 = 100

User receives 10 LP tokens more than justified, diluting the pool.

Tools Used

This issue was identified via manual review.

Recommendations

Explicitly call .reload()? on all SPL token accounts and mints immediately after any CPI that changes their state.

```
//In add_liquidity:
transfer_tokens(..., &ctx.accounts.vault_a, ...)?;
+ ctx.accounts.vault_a.reload()?;
transfer_tokens(..., &ctx.accounts.vault_b, ...)?;
+ ctx.accounts.vault_b.reload()?;
mint_to(cpi_ctx, lp_to_mint)?;
+ ctx.accounts.lp_mint.reload()?;

//In remove_liquidity:
burn(cpi_ctx_burn, lpt_to_redeem)?;
+ ctx.accounts.lp_mint.reload()?;
transfer_checked(..., &ctx.accounts.vault_a, ...)?;
+ ctx.accounts.vault_a.reload()?;
transfer_checked(..., &ctx.accounts.vault_b, ...)?;
+ ctx.accounts.vault_b.reload()?;
```

Critical Assumption of Equal Token Decimals Compromises Pool Integrity

Summary

The AMM contract does not validate whether token_mint_a and token_mint_b share the same number of decimals during pool initialization. Furthermore, key arithmetic operations that determine liquidity provisions and LP token minting assume both tokens use identical decimal scales. This design flaw allows users to create liquidity pools with mismatched token decimals, leading to incorrect ratio calculations, imbalanced reserves, and potential value extraction via arbitrage.

Vulnerability Details

In the initialize_pool function of liquidity_operations.rs, no check is performed to ensure that token_mint_a.decimals matches token_mint_b.decimals. Consequently, the internal accounting logic, which operates directly on raw u64 token amounts, assumes 1 unit of token A is equivalent to 1 unit of token B—an assumption that fails if the tokens have different decimal precision.

This mismatch propagates to two critical functions:

calculate_token_b_provision_with_a_given: This function computes how many units of token B must be added given an amount of token A, using the formula (reserve_b * amount_a) / reserve_a without any normalization to a common decimal scale.

calculate_lp_amount : It calculates the LP tokens to mint as sqrt(amount_a * amount_b) directly on raw u64 inputs, again ignoring the potential difference in token scales.

Proof of Concept

Token A: 6 decimals

Token B: 9 decimals

Initial reserves: 1 A = 1_000_000, 1 B = 1_000_000_000

User deposits 0.5 A (500_000 raw), contract computes 0.5 B (500_000_000 raw)

Despite appearing balanced, subsequent swaps drastically skew reserves due to the magnitude difference in decimals. This enables an attacker to extract liquidity by exploiting the inaccurate internal representation of value.

Impact

The absence of decimal normalization leads to misaligned liquidity ratios, breaking the invariant x * y = k that AMMs rely on. This results in:

Arbitrage opportunities due to distorted exchange rates

Value extraction at the expense of unsuspecting liquidity providers

Long-term pool imbalance and potential user fund losses

Tools Used

This issue was identified through manual code review.

Recommendations

Minimal Fix Enforce decimal equality during pool initialization:

```
require!(
  ctx.accounts.token_mint_a.decimals == ctx.accounts.token_mint_b.decimals
  AmmError::DecimalsMismatch
);
```

Optional Robust Fix Normalize all token values to a common internal scale (e.g., 18 decimals) before any arithmetic operations:

```
let normalized_reserve_a = (reserve_a as u128)
    .checked_mul(10u128.pow((18 - decimals_a) as u32))
    .unwrap();
let normalized_reserve_b = (reserve_b as u128)
    .checked_mul(10u128.pow((18 - decimals_b) as u32))
    .unwrap();
This ensures proportionality is preserved regardless of token mint configuration.
```

On-Chain Vaults Updated, Internal Reserves Left Stale

Summary

The swap_exact_in and swap_exact_out functions update the token vault balances on-chain but fail to synchronize these values back to the LiquidityPool account's reserve_a and reserve_b fields. This results in internal reserve values becoming stale and desynchronized from the real token balances held in the vaults. The inconsistency introduces downstream errors in liquidity operations and compromises the integrity of the pool's accounting logic.

Vulnerability Details

The AMM stores two types of reserve data:

Vault state: real-time token balances in token_vault_a.amount and token_vault_b.amount.

Logical state: on-chain fields liquidity_pool.reserve_a and liquidity_pool.reserve_b.

While swap functions read the correct values from the vaults during swap calculations:

let reserve_a: u64 = context.accounts.token_vault_a.amount; let reserve_b: u64 = context.accounts.token_vault_b.amount;

They fail to update the persistent LiquidityPool fields post-transaction. As a result, these fields retain outdated data.

Example scenario:

Initial state: vaults hold 100 A and 100 B; LiquidityPool.reserve_a = 100, reserve_b = 100.

After a swap of 10 A to B: vaults now hold 110 A and 91 B.

But LiquidityPool.reserve_a and reserve_b still report 100 A and 100 B.

This leads to inconsistencies in:

Liquidity withdrawals: LPs may request amounts based on outdated reserves.

Liquidity additions: skewed calculations of pool ratios and shares.

Off-chain metrics: external indexers or frontends reading LiquidityPool report false data.

Impact

Severity: High

Withdrawal logic may malfunction, returning excess tokens or reverting due to insufficient reserves.

Deposit/share calculations become inaccurate, breaking assumptions about proportional ownership.

Users and aggregators observing LiquidityPool fields will be misled about actual pool state.

Over time, desynchronization can cascade into severe accounting and trust failures.

Proof of Concept

Initial state:

- LiquidityPool.reserve_a = 100 - LiquidityPool.reserve_b = 100
- Vaults: 100 A, 100 B

Swap: 10 A → B

- New vaults:

110 A, 91 B

- LiquidityPool fields unchanged (still 100/100)

An LP withdraws assuming 100/100:

- Expects 100 A, 100 B - Only 91 B in vault → underflow or partial withdrawal

Steps to Reproduce

Deploy contract as-is.

Initialize a pool with 100 A and 100 B.

Perform swap_exact_in(10 A).

Read LiquidityPool.reserve_a/reserve_b → still 100/100.

Attempt full LP withdrawal or add liquidity based on pool state.

Observe incorrect results or transaction failure.

Recommendations

After each swap, immediately resynchronize the internal reserves:

```
+ context.accounts.liquidity_pool.reserve_a =
context.accounts.token_vault_a.amount;
+ context.accounts.liquidity_pool.reserve_b =
context.accounts.token_vault_b.amount;
```

This ensures that when Anchor serializes LiquidityPool, it reflects the actual vault balances and maintains accounting consistency across the protocol.