H1: `lock_pool` Operation Vulnerable to DoS

Description

The `lock_pool` operation requires creating a `lockEscrow` account to manage locked funds. However, the creation logic does not check for pre-existing accounts:

```
// Pseudocode for lock_pool (assumed based on description)
pub fn lock_pool(ctx: Context<LockPool>) -> Result<()> {
    // Creates lockEscrow account without checking if it exists
    let lock_escrow = ctx.accounts.lock_escrow.init()?;
    // ... (locking logic)
    Ok(())
}
```

A malicious actor can preemptively create the `lockEscrow` account, causing the `create_lock_escrow` transaction to fail and blocking the `lock_pool` operation.

Impact

This vulnerability enables an attacker to perform a Denial of Service (DoS) attack, preventing legitimate users from locking pools and disrupting the protocol's fundraising and liquidity processes.

Recommendation

Modify the `lock_pool` operation to check if the `lock_escrow` account exists before attempting creation, skipping the initialization if it already exists.

H2: Missing Update of `migration_token_allocation` in `Global` Struct

Description

The `Global::update_settings` function, used in the `set_params` instruction, fails to update the `migration_token_allocation` field in the `Global` struct:

```rust
// Pseudocode for update_settings (assumed based on description)
pub fn update_settings(ctx: Context<UpdateSettings>, input:
GlobalSettingsInput) -> Result<()> {
    let global = &mut ctx.accounts.global;
    // Updates other fields but omits migration_token_allocation
    global.some_field = input.some_field;
    // ... (no update for migration_token_allocation)
    Ok(())
}
```

This omission causes `migration_token_allocation`, used in the `create_pool` instruction, to remain at its default value, ignoring intended updates via `GlobalSettingsInput`.

Impact

The persistent incorrect `migration_token_allocation` value disrupts the migration process, potentially leading to misallocated tokens during pool creation and affecting the protocol's economic integrity.

Recommendation

Update the `Global::update_settings` function to include logic for modifying `migration_token_allocation` based on `GlobalSettingsInput`, and add unit tests to verify the update.

M1: Incorrect Fee Calculation in Last Buy

Description

In the "last buy" process, the protocol adjusts the transaction price to align with the bonding curve, altering the SOL amount paid by the user. However, the swap fee is calculated before this adjustment:

```rust
// Pseudocode for buy process (assumed based on description)
pub fn buy(ctx: Context<Buy>, buy_amount: u64) -> Result<()> {
    let fee_lamports = calculate_fee(buy_amount);
    let buy_result = apply_buy(ctx, buy_amount)?;
    // buy_result.exact_in_amount may differ from buy_amount_applied
    // No recheck of fee or user lamports
    transfer_sol(ctx.accounts.user, buy_result.exact_in_amount)?;
    Ok(())
}
```

This leads to incorrect fee calculations, and the user's lamport balance is not revalidated to ensure sufficient funds post-adjustment.

Impact

Incorrect fee calculations may result in under- or overcharging users, while insufficient balance checks could allow transactions to close accounts improperly, disrupting the protocol's financial accuracy.

Recommendation

After `apply_buy`, verify that the `buy_result.exact_in_amount` matches `buy_amount_applied`, recalculate `fee_lamports` if they differ, and revalidate `ctx.accounts.user.get_lamports() >= exact_in_amount.checked_add(min_rent).unwrap()`. Introduce a slippage parameter to control maximum SOL input.

M2: Abrupt Fee Transition at Slot 250

Description

The fee calculation in the bonding curve uses a linear decrease formula that causes an abrupt transition from 8.76% to 1% between slots 250 and 251:

```rust
// From bonding curve implementation
pub fn calculate_fee(&self, slot: u64) -> u64 {
    if slot <= 250 {
        // Linear decrease from 8.76% to incorrect value
        let fee_percent = 876 - ((slot as u128 * 776) / 250) as u64;
        // Results in discontinuity at slot 251
        (fee_percent * amount) / 10000
    } else {
        amount / 100 // 1% fee
    }
}
```

This discontinuity deviates from the intended smooth fee transition between Phase 2 and Phase 3.

Impact

The abrupt fee drop creates an economic discontinuity, potentially enabling arbitrage opportunities or user confusion, which could affect trust and participation in the protocol.

Recommendation

Recalibrate the linear decrease formula coefficients to ensure the fee reaches exactly 1% at slot 250, ensuring a smooth transition without discontinuities.