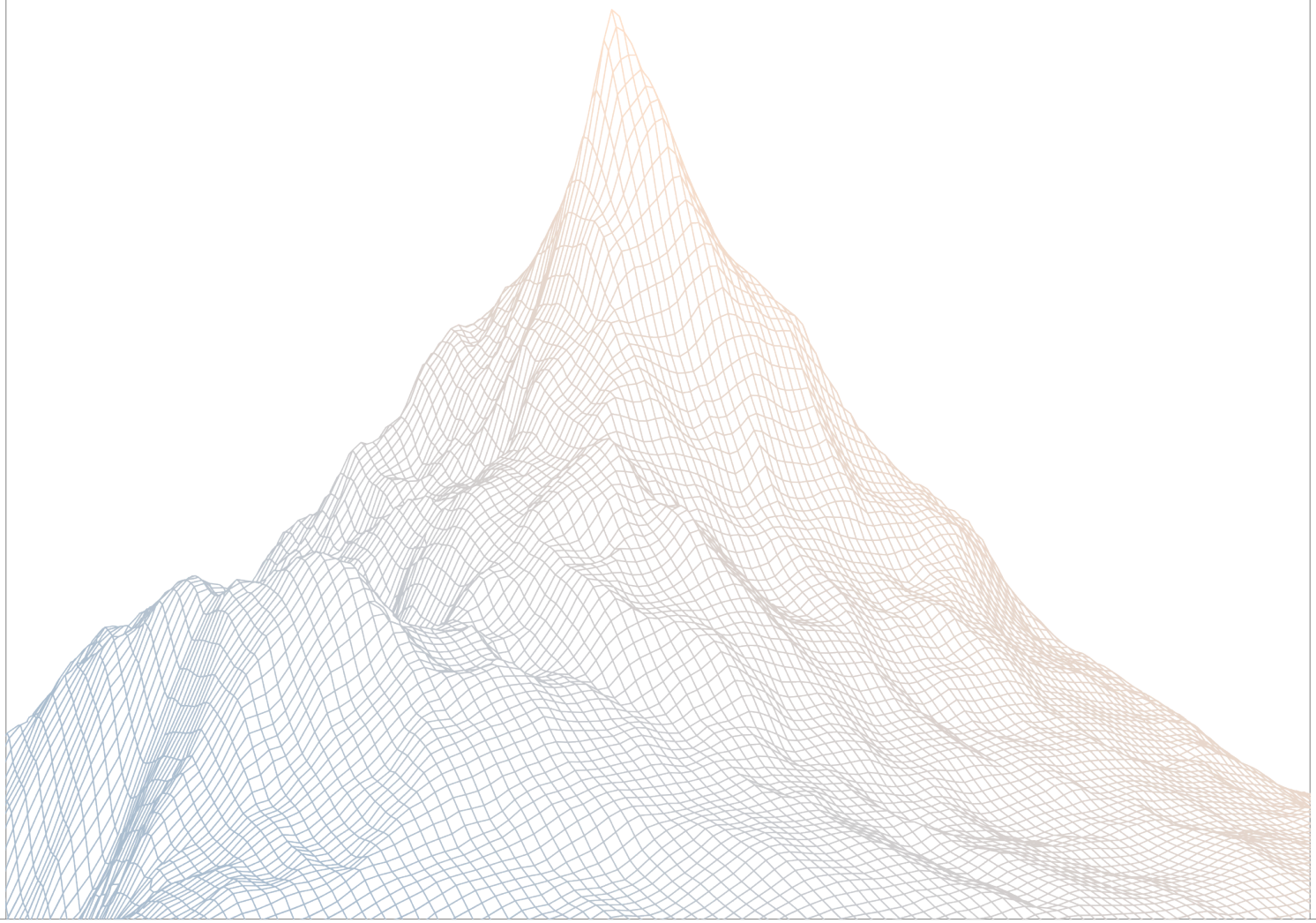
Zenith

Meteora

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: April 23rd to April 26th, 2025

AUDITED BY: J4X
peakbolt

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Meteora

Our mission is to build the most secure, sustainable and composable liquidity layer for all of Solana and DeFi.

By using Meteora's DLMM and Dynamic AMM Pools, liquidity providers can earn the best fees and yield on their capital.

This would help transform Solana into the ultimate trading hub for mainstream users in crypto by driving sustainable, long-term liquidity to the platform. Join us at Meteora to shape Solana's future as the go-to destination for all crypto participants.

2.2 Scope

The engagement involved a review of the following targets:

Target	dynamic-bonding-curve
Repository	https://github.com/MeteoraAg/dynamic-bonding-curve
Commit Hash	`186922825468d079018edddb4ff00323ee1e02e6`
Files	`Changes in PR #53`

2.3 Audit Timeline

April 23, 2025	Audit start
April 26, 2025	Audit end
May 05, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	0
Low Risk	4
Informational	3
Total Issues	**8**

3

Findings Summary

ID	Description	Status
H-1	Migration can be DoS as the curve completion check in swap() fails to account for locked vesting amount	Resolved
L-1	Config allows a high sqrt_start_price that leaves no buffer for swap_base_amount	Acknowledged
L-2	get_initial_liquidity_from_delta_base() could overflow and prevent migration	Resolved
L-3	Missing rent refund on DAMM creation	Acknowledged
L-4	get_migration_base_token() should round up for MigrationOption::DammV2	Acknowledged
I-1	escrow_token is missing documentation	Resolved
I-2	Consider adding explanation of post_migration_token_supply for fixed token supply pool	Resolved
I-3	Dust quote tokens could remain after migration_damm_v2()	Acknowledged

4

Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Migration can be DoS as the curve completion check in `swap()` fails to account for locked vesting amount

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- ix_swap.rs#L216-L223

Description:

When a `swap()` leads to a completed curve, it will perform a check to ensure that there are enough base tokens for the migration and fees as shown in the code below.

However, it fails to check that there are sufficient base tokens for the locked vesting, which will require transferring them to the locker.

An attacker can DoS the migration by performing a final swap for a significant amount of base tokens from the virtual pool. This will consume the base tokens that are meant for the locked vesting and then leave insufficient tokens for the migration. That prevents the migration, causing all the tokens to be stuck in the virtual pool.

```
pub fn handle_swap(ctx: Context<SwapCtx>, params: SwapParameters) →
    Result<()> {
    ..
    if pool.is_curve_complete(config.migration_quote_threshold) {
        ctx.accounts.base_vault.reload()?;
        // validate if base reserve is enough token for migration
        let base_vault_balance = ctx.accounts.base_vault.amount;

       //@audit this should include the total_locked_vesting_amount
        let required_base_balance = config
            .migration_base_threshold
            .safe_add(pool.get_protocol_and_partner_base_fee()?)?;
```

```
        require!(
            base_vault_balance ≥ required_base_balance,
            PoolError::InsufficientLiquidityForMigration
        );
```

Proof-of-Concept:

The following test shows that a final swap that buys out significant base tokens will cause `Migrate to Meteora Damm Pool` step to fail due to insufficient funds for migration.

```
import { BN } from "bn.js";
import { ProgramTestContext } from "solana-bankrun";
import {
  BaseFee,
  claimProtocolFee,
  ClaimTradeFeeParams,
  claimTradingFee,
  ConfigParameters,
  createClaimFeeOperator,
  createConfig,
  CreateConfigParams,
  createPoolWithSplToken,
  partnerWithdrawSurplus,
  protocolWithdrawSurplus,
  swap,
  SwapParams,
  //@audit import createLocker
  createLocker,
} from "./instructions";
import { Pool, VirtualCurveProgram } from "./utils/types";
import { Keypair, LAMPORTS_PER_SOL, PublicKey } from "@solana/web3.js";
import { deriveMetadatAccount, fundSol, getMint, startTest } from "./utils";
import {
  createDammConfig,
  createVirtualCurveProgram,
  derivePoolAuthority,
  MAX_SQRT_PRICE,
  MIN_SQRT_PRICE,
  U64_MAX,
} from "./utils";
import { getVirtualPool } from "./utils/fetcher";
import { NATIVE_MINT } from "@solana/spl-token";
import {
  createMeteoraMetadata,
  lockLpForCreatorDamm,
```

```
  lockLpForPartnerDamm,
  MigrateMeteoraParams,
  migrateToMeteoraDamm,
} from "./instructions/meteoraMigration";
import { assert, expect } from "chai";

describe("Full flow with spl-token", () => {
  let context: ProgramTestContext;
  let admin: Keypair;
  let operator: Keypair;
  let partner: Keypair;
  let user: Keypair;
  let poolCreator: Keypair;
  let program: VirtualCurveProgram;
  let config: PublicKey;
  let virtualPool: PublicKey;
  let virtualPoolState: Pool;
  let dammConfig: PublicKey;
  let claimFeeOperator: PublicKey;

  before(async () => {
    context = await startTest();
    admin = context.payer;
    operator = Keypair.generate();
    partner = Keypair.generate();
    user = Keypair.generate();
    poolCreator = Keypair.generate();
    const receivers = [
      operator.publicKey,
      partner.publicKey,
      user.publicKey,
      poolCreator.publicKey,
    ];
    await fundSol(context.banksClient, admin, receivers);
    await fundSol(context.banksClient, admin, receivers);
    program = createVirtualCurveProgram();
  });

  it.only("Admin create claim fee operator", async () => {
    claimFeeOperator = await createClaimFeeOperator(
      context.banksClient,
      program,
      {
        admin,
        operator: operator.publicKey,
      }
    );
```

```
});

it.only("Partner create config", async () => {
  const baseFee: BaseFee = {
    cliffFeeNumerator: new BN(2_500_000),
    numberOfPeriod: 0,
    reductionFactor: new BN(0),
    periodFrequency: new BN(0),
    feeSchedulerMode: 0,
  };

  const curves = [];

  for (let i = 1; i <= 16; i++) {
    if (i == 16) {
      curves.push({
        sqrtPrice: MAX_SQRT_PRICE,
        liquidity: U64_MAX.shln(30 + i),
      });
    } else {
      curves.push({
        sqrtPrice: MAX_SQRT_PRICE.muln(i * 5).divn(100),
        liquidity: U64_MAX.shln(30 + i),
      });
    }

  }

  const instructionParams: ConfigParameters = {
    poolFees: {
      baseFee,
      dynamicFee: null,
    },
    activationType: 0,
    collectFeeMode: 0,
    migrationOption: 0,
    tokenType: 0, // spl_token
    tokenDecimal: 6,
    migrationQuoteThreshold: new BN(LAMPORTS_PER_SOL * 5),
    partnerLpPercentage: 0,
    creatorLpPercentage: 0,
    partnerLockedLpPercentage: 95,
    creatorLockedLpPercentage: 5,
    sqrtStartPrice: MIN_SQRT_PRICE.shln(32),
    //@audit configure lock vesting
    /*
    lockedVesting: {
```

```
        amountPerPeriod: new BN(0),
        cliffDurationFromMigrationTime: new BN(0),
        frequency: new BN(0),
        numberOfPeriod: new BN(0),
        cliffUnlockAmount: new BN(0),
      },
      */
      lockedVesting: {
        amountPerPeriod: new BN(1_000_000),
        cliffDurationFromMigrationTime: new BN(0),
        frequency: new BN(1), // each 1 second
        numberOfPeriod: new BN(10),
        cliffUnlockAmount: new BN(1_000_000_000),
      },

      migrationFeeOption: 0,
      tokenSupply: null,
      padding: [],
      curve: curves,
    };
    const params: CreateConfigParams = {
      payer: partner,
      leftoverReceiver: partner.publicKey,
      feeClaimer: partner.publicKey,
      quoteMint: NATIVE_MINT,
      instructionParams,
    };
    config = await createConfig(context.banksClient, program, params);
  });

  it.only("Create spl pool from config", async () => {
    virtualPool = await createPoolWithSplToken(context.banksClient, program,
    {
      poolCreator,
      payer: operator,
      quoteMint: NATIVE_MINT,
      config,
      instructionParams: {
        name: "test token spl",
        symbol: "TEST",
        uri: "abc.com",
      },
    });
    virtualPoolState = await getVirtualPool(
      context.banksClient,
      program,
      virtualPool
```

```
  );

  // validate freeze authority
  const baseMintData = (
    await getMint(context.banksClient, virtualPoolState.baseMint)
  );
  expect(baseMintData.freezeAuthority.toString()).
  eq(PublicKey.default.toString())
  expect(baseMintData.mintAuthorityOption).eq(0)
});

it.only("Swap", async () => {
  const params: SwapParams = {
    config,
    payer: user,
    pool: virtualPool,
    inputTokenMint: NATIVE_MINT,
    outputTokenMint: virtualPoolState.baseMint,
    //amountIn: new BN(LAMPORTS_PER_SOL * 5.5),
    //@audit buy out the base tokens meant for locked vesting
    amountIn: new BN(LAMPORTS_PER_SOL * 15),
    minimumAmountOut: new BN(0),
    referralTokenAccount: null,
  };
  await swap(context.banksClient, program, params);
});

//@audit added create locker
it.only("Create locker", async () => {
    await createLocker(context.banksClient, program, {
        payer: admin,
        virtualPool,
    });
});

it.only("Create meteora metadata", async () => {
  await createMeteoraMetadata(context.banksClient, program, {
    payer: admin,
    virtualPool,
    config,
  });
});

it.only("Migrate to Meteora Damm Pool", async () => {
  const poolAuthority = derivePoolAuthority();
  dammConfig = await createDammConfig(
    context.banksClient,
```

```
      admin,
      poolAuthority
    );
    const migrationParams: MigrateMeteoraParams = {
      payer: admin,
      virtualPool,
      dammConfig,
    };

    //@audit this will fail due to insufficient funds
    await migrateToMeteoraDamm(context.banksClient, program,
    migrationParams);

    // validate mint authority
    const baseMintData = (
      await getMint(context.banksClient, virtualPoolState.baseMint)
    );
    expect(baseMintData.mintAuthorityOption).eq(0)
  });

});
```

Recommendations:

Include the total locked vesting amount in the `required_base_balance` for the curve completion check.

Meteora: Resolved with @07278373d7...

Zenith: Verified. Resolved by adding total locked vesting amount to `required_base_balance`.

4.2 Low Risk

A total of 4 low risk findings were identified.

[L-1] Config allows a high sqrt_start_price that leaves no buffer for swap_base_amount

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- ix_create_config.rs#L185-L189

Description:

`create_config()` allows `sqrt_start_price` to be between `MIN_SQRT_PRICE` (inclusive) and `MAX_SQRT_PRICE`.

```
// validate price and liquidity
require!(
    self.sqrt_start_price ≥ MIN_SQRT_PRICE && self.sqrt_start_price
    < MAX_SQRT_PRICE,
    PoolError::InvalidCurve
);
```

While `get_swap_amount_with_buffer()` will cap the `swap_amount_buffer()` to `max_base_amount_on_curve`.

That means it is possible to create a valid config with a `sqrt_start_price` that is high enough such that it does not have any buffer for the `swap_base_amount` due to the cap.

Without any buffer, it could slightly delay the curve completion as it now requires the last buyer to swap out an exact amount of base tokens that is available on the curve. The last buyer's swap could fail when an attacker try to grief it by just swapping out 1 unit of base token at a time. That will leave insufficient base tokens to fulfill the last buyer's swap as there are no more buffer. Despite that, the curve will eventually reach completion after some time as the last buyer can just retry with a smaller amount.

It is noted that the likelihood of this issue is low as partner is not expected to create such a

config with a high `sqrt_start_price`.

Recommendations:

Consider restricting `sqrt_start_price` such that it cannot be set to a high price.

Meteora: Acknowledged. This issue is mitigated by latest code that allows pool swallow surplus quote up to 20% of migration quote threshold.

Zenith: Mitigated by allowing pool to swallow the extra quote tokens beyond migration quote threshold. That means the last buyer can choose to pay more than the current price to complete the curve by setting a reasonable slippage value.

[L-2] get_initial_liquidity_from_delta_base() could overflow and prevent migration

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- curve.rs#L28-L42

Description:

`get_initial_liquidity_from_delta_base()` performs the compution `base_amount * sqrt_price * sqrt_max_price / price_delta` and then cast the result to u128 before returning it.

However, if the migration_sqrt_price is very high, this could potentially overflow u128 and cause it to fail.

Here is a example scenario that causes overflow

- `base_amount = 1,000,000e9 = 1e15` [1 mil base token in 9 decimals]
- `sqrt_price = 1e25` [migration_sqrt_price]
- `sqrt_max_price = 7.9227e28` [based on MAX_SQRT_PRICE]

That means the computation `base_amount * sqrt_price * sqrt_max_price / price_delta` could reach `(1e15 * 1e25* 1e28)/ 1e28 = 1e40`. And this will exceed `u128 (~3.4e38)` and cause an overflow.

If an overflow occurs, it will cause `get_liquidity_for_adding_liquidity()` to fail for `migrate_damm_v2()`. This will prevent migration from succeeding.

Though the likelihood of this issue occurring is low as this requires a high `migration_sqrt_price` and a large `migration_base_threshold`, which is not likely for standard quote tokens like SOL and USDC.

```
// □a = L * (1 / √P_lower - 1 / √P_upper) ⇒ L = □a / (1 / √P_lower - 1 /
    √P_upper)
pub fn get_initial_liquidity_from_delta_base(
    base_amount: u64,
    sqrt_max_price: u128,
    sqrt_price: u128,
```

```
) → Result<u128> {
    let price_delta = U512::from(sqrt_max_price.safe_sub(sqrt_price)?);
    let prod = U512::from(base_amount)
        .safe_mul(U512::from(sqrt_price))?
        .safe_mul(U512::from(sqrt_max_price))?;
    let liquidity = prod.safe_div(price_delta)?; // round down
    return Ok(liquidity
        .try_into()
        .map_err(|_| PoolError::TypeCastFailed)?);
}
```

Recommendations:

Despite the low likelihood, it is recommended to use at least u256 for result of `get_initial_liquidity_from_delta_base()`. Alternatively, consider validating the `get_liquidity_for_adding_liquidity()` during virtual pool creation.

Meteora: Resolved with @acefafa7ee...

Zenith: Verified. Resolved by using U512 for `get_initial_liquidity_from_delta_base()`.

[L-3] Missing rent refund on DAMM creation

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- /migration/dynamic_amm_v2/migrate_damm_v2_initialize_pool.rs#L185-L190
- /migration/create_locker.rs#L97

Description:

The `migrate_damm_v2_initialize_pool` IX charges the user 50 million lamports for rent for the creation of the pool,

```
invoke(
    &system_instruction::transfer(
        &self.payer.key(),
        &self.pool_authority.key(),
        50_000_000, // TODO calculate correct lamport here
    ),
    &[
        self.payer.to_account_info(),
        self.pool_authority.to_account_info(),
        self.system_program.to_account_info(),
    ],
)?;
```

Addditionally the `create_locker` IX, lamports are sent from the payer to the pool authority to cover the rent for the pool creation. However, this rent is overestimated to ensure that the pool can be deployed.

```
// Send some lamport to pool authority to pay rent fee?
msg!("transfer lamport to pool authority");
invoke(
    &system_instruction::transfer(
        &ctx.accounts.payer.key(),
        &ctx.accounts.pool_authority.key(),
        10_000_000, // TODO calculate correct lamport here
    ),
```

```
    &[
        ctx.accounts.payer.to_account_info(),
        ctx.accounts.pool_authority.to_account_info(),
        ctx.accounts.system_program.to_account_info(),
    ],
)?;
```

However, the overpayment is never refunded to the payer post creation, resulting in a loss for him.

Recommendations:

We recommend refunding the unused lamports to the payer.

Meteora: Acknowledged

[L-4] `get_migration_base_token()` should round up for `MigrationOption::DammV2`

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- liquidity_distribution.rs#L92-L97

Description:

`get_migration_base_token()` will round up the base token amount for `MigrationOption::MeteoraDamm`.

However, `MigrationOption::DammV2` does not follow the same design as it will round down the required `liquidity` in `get_initial_liquidity_from_delta_quote`.

```
MigrationOption::DammV2 => {

    //@audit this will round down and cause base_amount to be lower, and
    migration price will slightly higher
    let liquidity = get_initial_liquidity_from_delta_quote(
        migration_threshold,
        MIN_SQRT_PRICE,
        sqrt_migration_price,
    )?;

    // calculate base threshold
    let base_amount = get_delta_amount_base_unsigned_256(
        sqrt_migration_price,
        MAX_SQRT_PRICE,
        liquidity,
        Rounding::Up,
    )?;
```

Recommendations:

Consider rounding up the liquidity amount for `MigrationOption::DammV2` in `get_migration_base_token()`.

Meteora: Acknowledged.

Zenith: Rounding down is intended for `get_initial_liquidity_from_delta_quote()` as it is also used for `migrate_damm_v2_initialize_pool()` to calculate the liquidity for migration. Rounding up the liquidity amount will cause the calculated quote token amount for migration to exeed the `migration_quote_threshold`.

4.3 Informational

A total of 3 informational findings were identified.

[I-1] escrow_token is missing documentation

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Medium

Target

- programs/dynamic-bonding-curve/src/instructions/migration/create_locker.rs#L52-L54

Description:

The locker program will create an escrow PDA in the transaction. However, the ATA for the escrow is not created in that interaction.

```
/// Escrow.
#[account(
    init,
    seeds = [
        b"escrow".as_ref(),
        base.key().as_ref(),
    ],
    bump,
    payer = sender,
    space = 8 + VestingEscrow::INIT_SPACE
)]
pub escrow: AccountLoader<'info, VestingEscrow>,

// Mint.
pub token_mint: Box<InterfaceAccount<'info, Mint>>,

/// Escrow Token Account.
#[account(
    mut,
    associated_token::mint = token_mint,
    associated_token::authority = escrow,
```

```
    associated_token::token_program = token_program
)]
pub escrow_token: Box<InterfaceAccount<'info, TokenAccount>>,
```

As a result, the user migrating will need to pre-create the `escrow_token` ATA. However, this is never documented anywhere, and will result in the locker creation reverting if not done prior.

Recommendations:

We recommend either adding information that this ATA needs to be pre-created by the caller, or adding an `init_if_needed` to the account in the context.

Meteora: Resolved with PR-74

Zenith: Verified

[I-2] Consider adding explanation of post_migration_token_supply for fixed token supply pool

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- config.rs#L538-L552

Description:

Virtual pool can be configured as a fixed token supply pool by specifying `pre_migration_token_supply` and `post_migration_token_supply`.

One may assume that both `pre_migration_token_supply` and `post_migration_token_supply` specifies the hard cap on the token supply before and after migration, so that the token supply is fixed and do not exceed these caps.

However, that is not true as migration only burns `min(left_base_token, pre_migration_token_supply - post_migration_token_supply)`. That means it is possible to burn less than `pre_migration_token_supply - post_migration_token_supply`, when `left_base_token` is lower than that.

The final total supply will likely fall somewhere between `post_migration_token_supply` and `pre_migration_token_supply` .

```
fn get_max_burnable_amount_post_migration(&self) → Result<u64> {
    if self.is_fixed_token_supply() {
        Ok(self
            .pre_migration_token_supply
            .safe_sub(self.post_migration_token_supply)?)
    } else {
        Ok(u64::MAX)
    }
}

/// leftover is extra base token in base vault after curve is completed
pub fn get_burnable_amount_post_migration(&self, leftover: u64) →
    Result<u64> {
    let max_burnable_amount =
    self.get_max_burnable_amount_post_migration()?;
```

```
    Ok(max_burnable_amount.min(leftover))
}
```

Recommendations:

It will be helpful to add comments to explain that the final total supply will be between `post_migration_token_supply` and `pre_migration_token_supply` .

Meteora: Resolved with @220d25bf67...

Zenith: Resolved with comments for `post_migration_token_supply`.

[I-3] Dust quote tokens could remain after migration_damm_v2()

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- migrate_damm_v2_initialize_pool.rs#L434-L439

Description:

`migrate_damm_v2()` will calculate the base and quote tokens required for migration into the damm v2 pool based on the `migration_sqrt_price`.

To do that, It does an initial calculation of the `initial_liquidity` using `get_liquidity_for_adding_liquidity()`. That will then call both `get_initial_liquidity_from_delta_base()` and `get_initial_liquidity_from_delta_quote()` to get the minimum liquidity required.

The `initial_liquidity` amount is then used to derive the quote tokens and base tokens required for migration.

However, both `get_initial_liquidity_from_delta_base()` and `get_initial_liquidity_from_delta_quote()` will round down the liquidity amount. That means the `initial_liquidity` that is used to derive the quote tokens will be slightly lower than the `migration_quote_threshold` that was calculated during virtual initialization.

Due to this there could be leftover dust quote tokens after migration as migration will use up slightly less than `migration_quote_threshold`.

Recommendations:

Consider allowing withdrawal of these dust quote tokens under surplus withdrawal.

Meteora: Acknowledged.