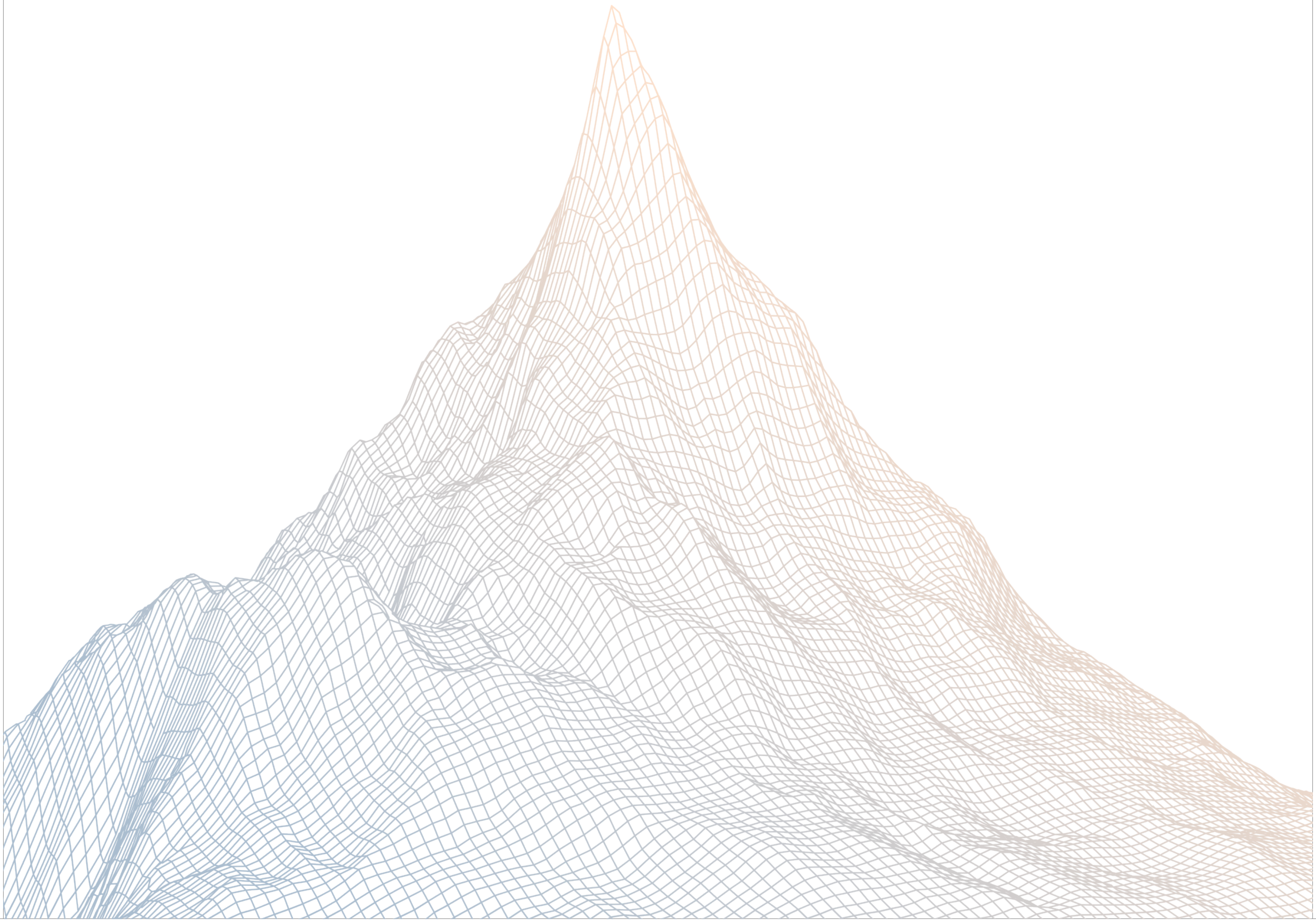
Zenith

Meteora DAMM v2

Smart Contract
Security Assessment

VERSION 1.1



AUDIT DATES:	May 26th to June 11th, 2025
AUDITED BY:	J4X peakbolt

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

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 DAMM v2

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	damm-v2
Repository	https://github.com/MeteoraAg/damm-v2
Commit Hash	`4f969508da67142265cafde1e2a14792b9000253`
Files	`programs/cp-amm/*`

2.3 Audit Timeline

May 26, 2025	Audit start
June 11, 2025	Audit end
June 17, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	4
Informational	8
Total Issues	**13**

3 Findings Summary

ID	Description	Status
M-1	Transfer fee is not considered on liquidity withdrawals	Resolved
L-1	SOL/USDC fee can't be claimed if token account was frozen	Resolved
L-2	Total Positions metric is incorrect	Resolved
L-3	Expenonential fee will be rounded down	Acknowledged
L-4	SOL/USDC fee can't be claimed if token account was frozen	Resolved
I-1	Users can be tricked to provide liquidity for high APY farm that cannot be claimed when frozen	Acknowledged
I-2	Missing check to prevent both reward indices from using the same reward mint	Acknowledged
I-3	0 Vesting is possible by not passing a cliff point	Resolved
I-4	Consider removing un-used codes	Resolved
I-5	Typo in comments	Resolved
I-6	Whitelisted tokens can never be removed	Resolved
I-7	total_claimed_reward could silently overflow	Acknowledged
I-8	Users can recover part of fee by referring themselves	Acknowledged

4 Findings

4.1 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] Transfer fee is not considered on liquidity withdrawals

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

Target

- cp-amm/src/instructions/ix_remove_liquidity.rs#L118-L126

Description:

Throughout the codebase, the slippage values are checked while considering a potential fee, which can be enforced through a Token2022 token. However, in the `handle_remove_liquidity`, this is not done, and the slippage is checked against the values pre-transfer.

```
let ModifyLiquidityResult {
    token_a_amount,
    token_b_amount,
} = pool.get_amounts_for_modify_liquidity(liquidity_delta, Rounding::Down)?;

require!(
    token_a_amount > 0 || token_b_amount > 0,
    PoolError::AmountIsZero
);
// Slippage check
require!(
    token_a_amount ≥ token_a_amount_threshold,
    PoolError::ExceededSlippage
);
require!(
    token_b_amount ≥ token_b_amount_threshold,
    PoolError::ExceededSlippage
);
```

POC:

```
describe("Token2022 with transfer fee", () ⇒ {
  let context: ProgramTestContext;
  let admin: Keypair;
  let user: Keypair;
  let creator: Keypair;
  let config: PublicKey;
  let pool: PublicKey;
  let position: PublicKey;
  let tokenAMint: PublicKey;
  let tokenBMint: PublicKey;

  beforeEach(async () ⇒ {
    const root = Keypair.generate();
    context = await startTest(root);

    user = await generateKpAndFund(context.banksClient, context.payer);
    admin = await generateKpAndFund(context.banksClient, context.payer);
    creator = await generateKpAndFund(context.banksClient, context.payer);

    // Create token A as regular SPL token
    tokenAMint = await createToken(
      context.banksClient,
      context.payer,
      context.payer.publicKey
    );

    // Create token B as Token2022 with 10% transfer fee
    tokenBMint = await createToken2022(
      context.banksClient,
      context.payer,
      [ExtensionType.TransferFeeConfig]
    );

    await mintSplTokenTo(
      context.banksClient,
      context.payer,
      tokenAMint,
      context.payer,
      user.publicKey
    );

    await mintToToken2022(
      context.banksClient,
      context.payer,
      tokenBMint,
      context.payer,
      user.publicKey
    );

    await mintSplTokenTo(
      context.banksClient,
      context.payer,
      tokenAMint,
      context.payer,
      creator.publicKey
    );

    await mintToToken2022(
      context.banksClient,
      context.payer,
      tokenBMint,
      context.payer,
      creator.publicKey
    );

    // Create config
    const createConfigParams = {
      poolFees: {
        baseFee: {
          cliffFeeNumerator: new BN(2_500_000),
          numberOfPeriod: 0,
          reductionFactor: new BN(0),
          periodFrequency: new BN(0),
          feeSchedulerMode: 0,
        },
        protocolFeePercent: 10,
        partnerFeePercent: 0,
        referralFeePercent: 0,
        dynamicFee: null,
      },
      sqrtMinPrice: new BN(MIN_SQRT_PRICE),
      sqrtMaxPrice: new BN(MAX_SQRT_PRICE),
      vaultConfigKey: PublicKey.default,
      poolCreatorAuthority: PublicKey.default,
      activationType: 0,
      collectFeeMode: 0,
    };

    config = await createConfigIx(
      context.banksClient,
      admin,
      new BN(randomID()),
      createConfigParams
    );

    // Initialize pool
    const initPoolParams = {
      payer: creator,
      creator: creator.publicKey,
      config,
      tokenAMint: tokenAMint,
      tokenBMint: tokenBMint,
      liquidity: new BN(MIN_LP_AMOUNT),
      sqrtPrice:
    new BN(MIN_SQRT_PRICE).add(new BN(MAX_SQRT_PRICE).div(new BN(2))),
      activationPoint: null,
    };

    const result = await initializePool(context.banksClient,
    initPoolParams);
    pool = result.pool;
  });

  it("User adds and removes liquidity with token2022 transfer fee", async ()
    ⇒ {
    // Create position
    const position = await createPosition(
      context.banksClient,
      user,
      user.publicKey,
      pool
    );

    // Get or create associated token accounts for user (declare once)
    const userTokenAAccount = await getOrCreateAssociatedTokenAccount(
      context.banksClient,
      context.payer,
      tokenAMint,
      user.publicKey,
      TOKEN_PROGRAM_ID
    );
    const userTokenBAccount = await getOrCreateAssociatedTokenAccount(
      context.banksClient,
      context.payer,
      tokenBMint,
      user.publicKey,
      TOKEN_2022_PROGRAM_ID
    );
```

```
// Log balances before addLiquidity
const userTokenABalanceBeforeAdd = AccountLayout.decode((await
context.banksClient.getAccount(userTokenAAccount)).data).amount;
const userTokenBBalanceBeforeAdd = AccountLayout.decode((await
context.banksClient.getAccount(userTokenBAccount)).data).amount;

// Add liquidity
let liquidity = new BN("100000000000");
const addLiquidityParams = {
  owner: user,
  pool,
  position,
  liquidityDelta: liquidity,
  tokenAAmountThreshold: U64_MAX,
  tokenBAmountThreshold: U64_MAX,
};
await addLiquidity(context.banksClient, addLiquidityParams);

// Log balances after addLiquidity
const userTokenABalanceAfterAdd = AccountLayout.decode((await
context.banksClient.getAccount(userTokenAAccount)).data).amount;
const userTokenBBalanceAfterAdd = AccountLayout.decode((await
context.banksClient.getAccount(userTokenBAccount)).data).amount;

// Calculate and log the amount spent by the user
const tokenASpent
= new BN(userTokenABalanceBeforeAdd.toString()).sub(new
BN(userTokenABalanceAfterAdd.toString()));
const tokenBSpent
= new BN(userTokenBBalanceBeforeAdd.toString()).sub(new
BN(userTokenBBalanceAfterAdd.toString()));
console.log("Token A spent by user for addLiquidity:",
tokenASpent.toString());
console.log("Token B spent by user for addLiquidity:",
tokenBSpent.toString());

// Set slippage to 95% of the amount the user put in
const slippageA = tokenASpent.mul(new BN(85)).div(new BN(100));
const slippageB = tokenBSpent.mul(new BN(85)).div(new BN(100));
console.log("Slippage threshold for Token A (85%):",
slippageA.toString());
console.log("Slippage threshold for Token B (85%):",
slippageB.toString());

// Remove liquidity
const removeLiquidityParams = {
      owner: user,
      pool,
      position,
      liquidityDelta: liquidity,
      tokenAAmountThreshold: slippageA,
      tokenBAmountThreshold: slippageB,
    };
    await removeLiquidity(context.banksClient, removeLiquidityParams);

    // Get user's token account balances after removal
    const userTokenABalanceAfter = AccountLayout.decode((await
    context.banksClient.getAccount(userTokenAAccount)).data).amount;
    const userTokenBBalanceAfter = AccountLayout.decode((await
    context.banksClient.getAccount(userTokenBAccount)).data).amount;

    // Calculate and log the difference
    const tokenAReceived
    = new BN(userTokenABalanceAfter.toString()).sub(new
    BN(userTokenABalanceAfterAdd.toString()));
    const tokenBReceived
    = new BN(userTokenBBalanceAfter.toString()).sub(new
    BN(userTokenBBalanceAfterAdd.toString()));

    console.log("Token A actually received by user:",
    tokenAReceived.toString());
    console.log("Token B actually received by user:",
    tokenBReceived.toString());
  });
});
```

Recommendations:

We recommend using `calculate_transfer_fee_excluded_amount()` to get the exact value.

Meteora: Resolved with PR-51

Zenith: Verified

4.2 Low Risk

A total of 4 low risk findings were identified.

[L-1] SOL/USDC fee can't be claimed if token account was frozen

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

Target

- cp-amm/src/instructions/partner/ix_claim_partner_fee.rs#L67-L85
- cp-amm/src/instructions/admin/ix_claim_protocol_fee.rs#L74-L92

Description:

On mint accounts, a freeze authority can be set. This authority can then freeze any token account holding tokens of this mint. A malicious owner could use this to freeze the token vault corresponding to the pool. This way neither the partner nor the protocol could withdraw their fees that were collected in the non quote token anymore.

```
/// Partner claim fees.
pub fn handle_claim_partner_fee(
    ctx: Context<ClaimPartnerFeesCtx>,
    max_amount_a: u64,
    max_amount_b: u64,
) → Result<()> {
    let mut pool = ctx.accounts.pool.load_mut()?;
    let (token_a_amount, token_b_amount)
    = pool.claim_partner_fee(max_amount_a, max_amount_b)?;

    transfer_from_pool(
        ctx.accounts.pool_authority.to_account_info(),
        &ctx.accounts.token_a_mint,
        &ctx.accounts.token_a_vault,
        &ctx.accounts.token_a_account,
        &ctx.accounts.token_a_program,
        token_a_amount,
        ctx.bumps.pool_authority,
    )?;

    transfer_from_pool(
        ctx.accounts.pool_authority.to_account_info(),
        &ctx.accounts.token_b_mint,
        &ctx.accounts.token_b_vault,
        &ctx.accounts.token_b_account,
        &ctx.accounts.token_b_program,
        token_b_amount,
        ctx.bumps.pool_authority,
    )?;

    emit_cpi!(EvtClaimPartnerFee {
        pool: ctx.accounts.pool.key(),
        token_a_amount,
        token_b_amount
    });
    Ok(())
}
```

The problem is that both transfers (quote token and non quote token) are in one call so if one of them fails the other one will also not be possible.

Recommendations:

We recommend using something like a try-catch block to still allow for withdrawing the quote tokens.

Meteora: Resolved with PR-60

Zenith: Verified

[L-2] Total Positions metric is incorrect

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

Target

- cp-amm/src/state/pool.rs#L168-L171

Description:

The `total_position` value of the `PoolMetrics` is used to track the total active positions in the pool. On creation of a new position, it is correctly incremented using `inc_position`.

```
pub fn initialize(
    &mut self,
    pool_state: &mut Pool,
    pool: Pubkey,
    nft_mint: Pubkey,
    liquidity: u128,
) → Result<()> {
    pool_state.metrics.inc_position()?;
    self.pool = pool;
    self.nft_mint = nft_mint;
    self.unlocked_liquidity = liquidity;
    Ok(())
}
```

However, when a position is closed using the `ix_close_position`, the corresponding `rec_position()` function is never called. Thus, the metrics will always display an inflated number of positions.

Recommendations:

We recommend calling the `rec_position()` function on closing a position.

Meteora: Resolved with PR-62

Zenith: Verified

[L-3] Expenonential fee will be rounded down

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

Target

- cp-amm/src/math/fee_math.rs#L28-L33

Description:

The base fee numerator for exponential fees is calculated by using the following formula:

$$cliffFeeNumerator * {(1-reductionFactor/10000)}^{passedPeriod}$$

It's implemented as follows:

```
let (fee, _) = result
    .safe_mul(cliff_fee_numerator.into())?
    .overflowing_shr(SCALE_OFFSET);
let fee_numerator = u64::try_from(fee).map_err(|_|
    PoolError::TypeCastFailed)?;
Ok(fee_numerator)
```

Due to the multiplication and the following shift right, it'll round the effective numerator down.

Recommendations:

We recommend rounding up if there is rounding.

```
let fee_numerator = u64::try_from(fee).map_err(|_|
    PoolError::TypeCastFailed)?;
if rounding == 1 {
    return Ok(fee_numerator + 1);}
else {
    return Ok(fee_numerator); }
```

Meteora: Acknowledged

[L-4] SOL/USDC fee can't be claimed if token account was frozen

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

Target

- cp-amm/src/instructions/partner/ix_claim_partner_fee.rs#L67-L85
- cp-amm/src/instructions/admin/ix_claim_protocol_fee.rs#L74-L92

Description:

On mint accounts, a freeze authority can be set. This authority can then freeze any token account holding tokens of this mint. A malicious owner could use this to freeze the token vault corresponding to the pool. This way neither the partner nor the protocol could withdraw their fees that were collected in the non quote token anymore.

```
/// Partner claim fees.
pub fn handle_claim_partner_fee(
    ctx: Context<ClaimPartnerFeesCtx>,
    max_amount_a: u64,
    max_amount_b: u64,
) → Result<()> {
    let mut pool = ctx.accounts.pool.load_mut()?;
    let (token_a_amount, token_b_amount)
    = pool.claim_partner_fee(max_amount_a, max_amount_b)?;

    transfer_from_pool(
        ctx.accounts.pool_authority.to_account_info(),
        &ctx.accounts.token_a_mint,
        &ctx.accounts.token_a_vault,
        &ctx.accounts.token_a_account,
        &ctx.accounts.token_a_program,
        token_a_amount,
        ctx.bumps.pool_authority,
    )?;

    transfer_from_pool(
        ctx.accounts.pool_authority.to_account_info(),
        &ctx.accounts.token_b_mint,
        &ctx.accounts.token_b_vault,
        &ctx.accounts.token_b_account,
        &ctx.accounts.token_b_program,
        token_b_amount,
        ctx.bumps.pool_authority,
    )?;

    emit_cpi!(EvtClaimPartnerFee {
        pool: ctx.accounts.pool.key(),
        token_a_amount,
        token_b_amount
    });
    Ok(())
}
```

The problem is that both transfers (quote token and non quote token) are in one call so if one of them fails the other one will also not be possible.

Recommendations:

We recommend using something like a try-catch block to still allow for withdrawing the quote tokens.

Meteora: Resolved with PR-60

Zenith: Verified

4.3 Informational

A total of 8 informational findings were identified.

[I-1] Users can be tricked to provide liquidity for high APY farm that cannot be claimed when frozen

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

Target

- ix_initialize_reward.rs#L51-L66

Description:

`initialize_rewards()` allows pools to be setup with farming rewards using any supported or whitelisted mint.

The pool owner (as a funder) can use `fund_rewards()` to configure the reward amount and duration. Note that it is not possible to change the reward duration, to ensure that the reward campaign is not disrupted once it has been funded.

```
// only allow update reward duration if previous reward has been finished
require!(
    reward_info.reward_duration_end < (current_time as u64),
    PoolError::RewardCampaignInProgress
);
```

However, `initialize_rewards()` does not restrict one from using a freezable token mint for farming reward. This allows the pool owner to be able to DoS the claiming of farm rewards and disrupt the reward campaign.

In the worst case, It is possible to abuse this issue to deceive users with high APR farm to buy base tokens and pump up the price, and then DoS the reward claiming.

Though the likelihood is low as users are expected to do their due diligence on the reward token mint.

Recommendations:

As it is stated in the docs that the process could be permissionless at a later stage, it is suggested to ensure the reward token do not have freeze authority (unless whitelisted or is the pool's tokens) to prevent abuse of it in the future.

Meteora: We acknowledge the issue. The reason is that most of the well-known token such as EURC, USDC have freeze authority. We want the token team can setup the farm permissionlessly.

[I-2] Missing check to prevent both reward indices from using the same reward mint

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

Target

- ix_initialize_reward.rs#L51-L66

Description:

`initialize_reward()` is used to initialize the pool's farming reward token based on the provided `reward_index`. Each pool are allowed to have at most 2 reward token mints (e.g. `reward_index` can only be 0 or 1).

However, `initialize_reward()` fails to ensure that both `reward_index` in the same pool do not have the same reward token mint.

Though this is unlikely to happen, it is still recommended to perform this safety check, as the reward token mint cannot be changed once it is initialized.

Recommendations:

Consider adding a check to prevent the same reward token mint to be used for both `reward_index` in the same pool.

Meteora: The future permissionless LM feature will allow only index 0 for pool creator, while the rest of the index will be reserved for admin. We wish to allow maximum flexibility for admin reserved indexes. Therefore, it's impossible to have duplicated mint for permissionless index.

Zenith: Acknowledged.

[I-3] 0 Vesting is possible by not passing a cliff point

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

Target

- cp-amm/src/instructions/ix_lock_position.rs#L116

Description:

The vestings are not intended to be of 0 value. To ensure this, the following check is implemented:

```
require!(
    self.get_total_lock_amount()? > 0,
    PoolError::InvalidVestingInfo
);
```

However, there still is a way to create a vesting that does not vest at all. To do this, a user can create a vesting with a `cliff_unlock_liquidity > 0` and all other values set to 0. Then the user doesn't pass a `cliff_point`, so it will be set to the current time. As a result, the check will pass, but the vesting is not vested at all, as all of it can be instantly unlocked again (even in the same transaction).

Recommendations:

We recommend checking if `number_of_period > 0` if no cliff point was passed.

Meteora: Resolved with PR-66

Zenith: Verified

[I-4] Consider removing un-used codes

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

Target

- fee_parameters.rs#L317-L347
- activation_handler.rs#L104-L110
- activation_handler.rs#L83-L86
- constants.rs#L103
- config.rs#L306
- activation_handler.rs#L129-L156

Recommendations:

Consider removing the following un-used code for readability and ease of maintenance.

1. Within `fee_parameters.rs`, both `accrue_partner_fees()` and `claim_fees()` are currently unused as the partner fee account is now stored and managed in `pool.rs`.

```
pub fn accrue_partner_fees(
    &mut self,
    protocol_fee: u64,
    trade_direction: TradeDirection,
) -> Result<()> {
    if self.fee_percent > 0 {
        let partner_profit = protocol_fee
            .safe_mul(self.fee_percent.into())?
            .safe_div(100)?;

        match trade_direction {
            TradeDirection::AtoB => {
                self.pending_fee_a
    = self.pending_fee_a.safe_add(partner_profit)?;
            }
            TradeDirection::BtoA => {
                self.pending_fee_b
    = self.pending_fee_b.safe_add(partner_profit)?;
```

```
                }
            }
        }
        Ok(())
    }

    pub fn claim_fees(&mut self, max_amount_a: u64, max_amount_b: u64) →
        Result<(u64, u64)> {
        let claimable_amount_a = max_amount_a.min(self.pending_fee_a);
        let claimable_amount_b = max_amount_b.min(self.pending_fee_b);

        self.pending_fee_a = self.pending_fee_a.safe_sub(claimable_amount_a)?;
        self.pending_fee_b = self.pending_fee_b.safe_sub(claimable_amount_b)?;

        Ok((claimable_amount_a, claimable_amount_b))
    }
```

2. Within `activation_handlers.rs`, the `validate_remove_balanced_liquidity()` is not used by any part of the program. Also `validate_swap()` and `get_last_buying_point()` are not used as swap current point validation is performed by `can_swap()` in `perrmissionless.rs`.

```
pub fn validate_remove_balanced_liquidity(&self) → Result<()> {
    require!(
        self.curr_point ⩾ self.activation_point,
        PoolError::PoolDisabled
    );
    Ok(())
}

pub fn validate_swap(&self, sender: Pubkey) → Result<()> {
    if sender == self.whitelisted_vault {
        require!(
            self.is_launch_pool()
                && self.curr_point ⩾ self.get_pre_activation_start_point()?
                && self.curr_point ⩽ self.get_last_buying_point()?,
            PoolError::PoolDisabled
        );
    } else {
        require!(
            self.curr_point ⩾ self.activation_point,
            PoolError::PoolDisabled
        );
    }
    Ok(())
}

pub fn get_last_buying_point(&self) → Result<u64> {
    let last_buying_slot = self.activation_point.safe_sub(1)?;
    Ok(last_buying_slot)
}
```

3. Within `constant.rs` there is an unused `MEME_MIN_FEE_UPDATE_WINDOW_DURATION` constant.

```
pub const MEME_MIN_FEE_UPDATE_WINDOW_DURATION: i64 = 60 * 30; // 30 minutes
```

4. The config.rs file includes the `to_bootstrapping_config()` function, which is not used anywhere in the codebase. We recommend removing the function, as it is not in use.

```
pub fn to_bootstrapping_config(&self, activation_point: u64) →
    BootstrappingConfig {
    BootstrappingConfig {
        activation_point,
        vault_config_key: self.vault_config_key,
        activation_type: self.activation_type,
    }
}
```

5. `validate_update_activation_point()` is un-reachable.

```
pub fn validate_update_activation_point(&self, new_activation_point: u64) →
    Result<()> {
    let nearest_new_activation_point =
    self.curr_point.safe_add(self.buffer_duration)?;
    require!(
        new_activation_point > nearest_new_activation_point
            && self.activation_point > self.curr_point,
        PoolError::UnableToModifyActivationPoint
    );

    if self.is_launch_pool() {
        // Don't allow update when the pool already enter pre-activation
    phase
        require!(
            self.curr_point < self.get_pre_activation_start_point()?,
            PoolError::UnableToModifyActivationPoint
        );

        let new_pre_activation_start_point =
            new_activation_point.safe_sub(self.buffer_duration)?;
        let buffered_new_pre_activation_start_point =
            new_pre_activation_start_point.safe_sub(self.buffer_duration)?;

        // Prevent update of activation point causes the pool enter
    pre-activation phase immediately, no time buffer for any correction as
    the crank will swap it
        require!(
            self.curr_point < buffered_new_pre_activation_start_point,
            PoolError::UnableToModifyActivationPoint
        );
    }

    Ok(())
}
```

Meteora: Removed un-used codes in PR-65.

Zenith: Verified

[I-5] Typo in comments

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

Target

- cp-amm/src/instructions/initialize_pool/ix_initialize_customizable_pool.rs#L347

Description:

The code comments include a typo.

```
// require at least 1 lamport to prove onwership of token mints
```

Recommendations:

We recommend fixing the typo.

Meteora: Resolved with PR-64

Zenith: Verified

[I-6] Whitelisted tokens can never be removed

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

Target

- cp-amm/src/instructions/admin/ix_create_token_badge.rs#L11

Description:

The `TokenBadge` account allows the admin to additionally whitelist token2022 tokens, including those not supported by the extensions. However, once these are whitelisted (by creating a `TokenBadge` account), the account cannot be closed so that it will remain whitelisted indefinitely.

This leads to issues if they were whitelisted due to having non-malicious code, but are later changed to malicious or non-compliant code. For example, a token with a disabled transfer hook could be whitelisted. Later on, the hook gets reactivated, leading to further problems.

Recommendations:

We recommend adding a function that allows for the closing of token badges.

Meteora: Fixed in PR-59

Zenith: Verified

[I-7] total_claimed_reward could silently overflow

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

Target

- cp-amm/src/state/position.rs#L233

Description:

The `accumulate_total_claimed_rewards()` function uses `wrapping_add` to increase the rewards a position has accrued.

```
fn accumulate_total_claimed_rewards(&mut self, reward_index: usize, reward:
    u64) {
    let total_claimed_reward =
    self.reward_infos[reward_index].total_claimed_rewards;
    self.reward_infos[reward_index].total_claimed_rewards =
        total_claimed_reward.wrapping_add(reward);
}
```

However, this function can silently overflow.

Recommendations:

We recommend using `safe_add` instead.

Meteora: Acknowledged.

[I-8] Users can recover part of fee by referring themselves

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

Target

- cp-amm/src/instructions/ix_swap.rs#L176-L199

Description:

The `swap` IX allows users to pass a referral account.

```
/// referral token account
#[account(mut)]
pub referral_token_account: Option<Box<InterfaceAccount<'info,
    TokenAccount>>>,
```

A part of the invoked fee will be sent to this account.

```
// send to referral
if has_referral {
    if fee_mode.fees_on_token_a {
        transfer_from_pool(
            ctx.accounts.pool_authority.to_account_info(),
            &ctx.accounts.token_a_mint,
            &ctx.accounts.token_a_vault,
            &ctx.accounts.referral_token_account.clone().unwrap(),
            &ctx.accounts.token_a_program,
            swap_result.referral_fee,
            ctx.bumps.pool_authority,
        )?;
    } else {
        transfer_from_pool(
            ctx.accounts.pool_authority.to_account_info(),
            &ctx.accounts.token_b_mint,
            &ctx.accounts.token_b_vault,
            &ctx.accounts.referral_token_account.clone().unwrap(),
            &ctx.accounts.token_b_program,
            swap_result.referral_fee,
            ctx.bumps.pool_authority,
        )?;
    }
}
```

However there is currently no restriction on this account. Thus a user can pass his own account and recover part of the fee.

POC:

```
it("User swap A→B with self as referrer", async () ⇒ {
  const addLiquidityParams: AddLiquidityParams = {
    owner: user,
    pool,
    position,
    liquidityDelta: new BN(MIN_SQRT_PRICE.muln(30)),
    tokenAAmountThreshold: new BN(200),
    tokenBAmountThreshold: new BN(200),
  };
  await addLiquidity(context.banksClient, addLiquidityParams);

  // Get the associated token account for the user
  const userTokenAccount = await getOrCreateAssociatedTokenAccount(
    context.banksClient,
    context.payer,
    outputTokenMint,
    user.publicKey,
    TOKEN_2022_PROGRAM_ID
  );

  const swapParams: SwapParams = {
    payer: user,
    pool,
    inputTokenMint,
    outputTokenMint,
    amountIn: new BN(10),
    minimumAmountOut: new BN(0),
    referralTokenAccount: userTokenAccount,
  };

  await swap(context.banksClient, swapParams);
});
```

Recommendations:

We recommend implementing a whitelist for trusted referrers and aborting if the `referral_token_account` is not one of those.

Meteora: Acknowledged.