MeteoraAg CPMM

Security Assessment

March 18th, 2025 — Prepared by OtterSec

Xiang Yin	soreatu@osec.io
Gabriel Ottoboni	ottoboni@osec.io
Kevin Chow	kchow@osec.io
Robert Chen	r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

MeteoraAg engaged OtterSec to assess the `cp-amm` program. This assessment was conducted between March 6th and March 15th, 2025. For more information on our auditing methodology, refer to Appendix C.

Key Findings

We produced 7 findings throughout this audit engagement.

In particular, we identified a vulnerability where it is possible for anyone to create a pool for any token pair due to a lack of token ownership check. This may be exploited to block legitimate creators by preemptively creating a pool for a particular token combination (OS-MCM-ADV-00).

We made recommendations for modifying the codebase for improved efficiency, functionality, and robustness (OS-MCM-SUG-03), and suggested removing redundant or unutilized code instances (OS-MCM-SUG-04). Additionally, we advised aborting the remove liquidity instruction when a user requests to remove more liquidity than they own (OS-MCM-SUG-01).

Scope

The source code was delivered to us in a Git repository at https://github.com/MeteoraAg/cp-amm. This audit was performed against commit 71f455a.

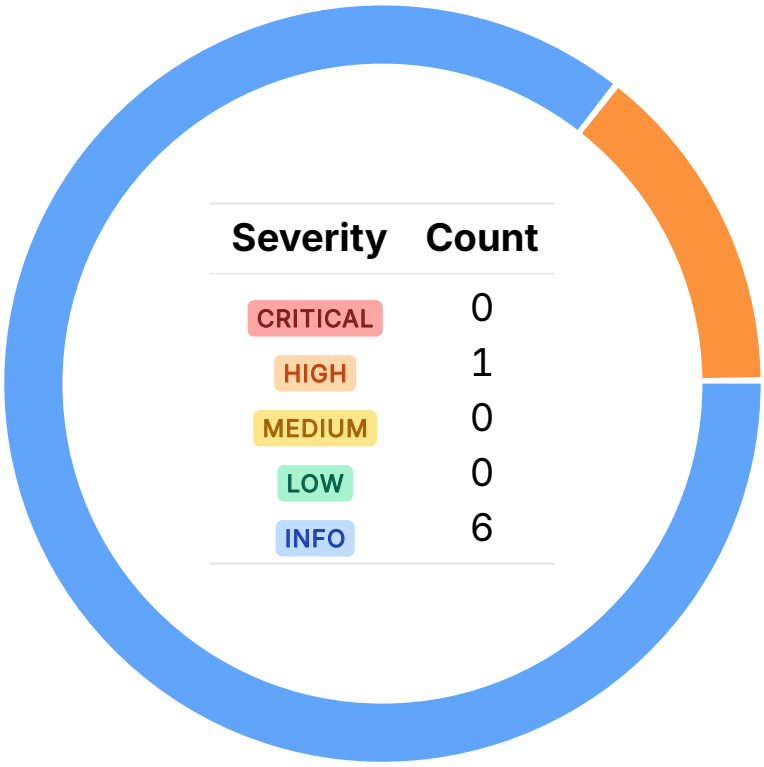
A brief description of the program is as follows:

Name	Description
cp-amm	A constant product market maker protocol that facilitates automated market-making and liquidity provision for token pairs.

02 — Findings

Overall, we reported 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	0
LOW	0
INFO	6

03 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix B.

ID	Severity	Status	Description
OS-MCM-ADV-00	HIGH	RESOLVED	The customizable pool initialization lacks a check for token ownership, allowing anyone to create a pool for any token pair. This may be exploited to block legitimate creators by preemptively creating a pool for a particular token combination.

DOS due to Improper Pool Initialization HIGH OS-MCM-ADV-00

Description

The `ix_initialize_customizable_pool` instruction allows any user to create a customizable pool for a unique token pair, with only one pool permitted per pair. However, it does not validate whether the creator actually holds the required tokens. This omission may be exploited for a denial-of-service attack, where a user creates a pool without owning the tokens, thereby preventing legitimate users from creating the same pool pair.

>_ cp-mm/src/instructions/initialize_pool/ix_initialize_customizable_pool.rs RUST

```rust
pub struct InitializeCustomizablePoolCtx<'info> {
    [...]
    /// Initialize an account to store the pool state
    #[account(
        init,
        seeds = [
            CUSTOMIZABLE_POOL_PREFIX.as_ref(),
            &max_key(&token_a_mint.key(), &token_b_mint.key()),
            &min_key(&token_a_mint.key(), &token_b_mint.key()),
        ],
        bump,
        payer = payer,
        space = 8 + Pool::INIT_SPACE
    )]
    pub pool: AccountLoader<'info, Pool>,
    [...]
}
```

Remediation

Add a check to verify that the creator holds a non-zero balance of the tokens required to initialize the customizable pool.

Patch

Resolved in 5fe541d.

04 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-MCM-SUG-00	The proof incorrectly assumes that y is constant during a swap, whereas liquidity (L) is actually constant. Additionally, the derivation of L' as $L+x*\sqrt{P}$ is flawed.
OS-MCM-SUG-01	The current implementation caps the liquidity to be removed to the user's available amount, which may result in unexpected behavior for other programs when they call this instruction via CPI.
OS-MCM-SUG-02	The slippage protection should account for transfer fees when validating the `minimum_amount_out` after a swap to ensure users receive at least the expected amount, factoring in any fees.
OS-MCM-SUG-03	Recommendation for modifying the codebase for improved efficiency, functionality, and robustness.
OS-MCM-SUG-04	The codebase contains multiple cases of redundant or unutilized code that should be removed.
OS-MCM-SUG-05	Modifications to the codebase to rectify informational and typographical errors in the comments for improved code clarity.

Improper Proof Derivation

OS-MCM-SUG-00

Description

In `curve::get_next_sqrt_price_from_amount_a_rounding_up`, the function's inline proof incorrectly assumes constant y during a swap, which is not correct. Instead, liquidity (L) remains constant while token balances (`x` and `y`) change. Thus, it is not possible to derive L' as $L + \Delta x * \sqrt{P}$ correctly, because: $L + \Delta x * \sqrt{P} \Rightarrow x * \sqrt{P} + \Delta x * \sqrt{P} \Rightarrow x' * \sqrt{P} \neq x' * \sqrt{P'} = L'$.

>_ *programs/cp-amm/src/curve.rs* RUST

```rust
/// # Formula
///
/// * `√P' = √P * L / (L + Δx * √P)`
/// * If Δx * √P overflows, use alternate form `√P' = L / (L/√P + Δx)`
///
/// # Proof
///
/// For constant y,
/// √P * L = y
/// √P' * L' = √P * L
/// √P' = √P * L / L'
/// √P' = √P * L / L'
/// √P' = √P * L / (L + Δx*√P)
///
pub fn get_next_sqrt_price_from_amount_a_rounding_up([...])
```

Remediation

Ensure to utilize the correct assumption while performing the derivation. A derivation that results in $\sqrt{P'} = \sqrt{P} * L/(L + \Delta x * \sqrt{P})$ is attached for reference in Appendix A.

Failure to Abort on Insufficient Liquidity OS-MCM-SUG-01

Description

The current implementation of `ix_remove_liquidity::handle_remove_liquidity` caps the liquidity to be removed by calculating the minimum between `max_liquidity_delta` (requested removal) and `position.unlocked_liquidity` (actual available liquidity). This implies that if a user requests to remove more liquidity than they own, the function will silently adjust the removal amount instead of failing outright. This may result in unexpected behaviors, as other programs that call this instruction via CPI (Cross-Program Invocation) may assume that the requested `max_liquidity_delta` will be removed exactly.

>_ programs/cp-amm/src/instructions/ix_remove_liquidity.rs RUST

```rust
pub fn handle_remove_liquidity(
    ctx: Context<RemoveLiquidityCtx>,
    params: RemoveLiquidityParameters,
) -> Result<()> {
    [...]
    let liquidity_delta = position.unlocked_liquidity.min(max_liquidity_delta);
    [...]
}
```

Remediation

Modify the instruction such that it aborts when a user requests to remove more liquidity than they own, similar to what is done in Uniswap V3.

Fee-Adjusted Slippage Protection OS-MCM-SUG-02

Description

In the existing code within `ix_swap::handle_swap`, the check for slippage only compares the `swap_result.output_amount` (the amount of output tokens the user is expected to receive) to the `minimum_amount_out` (the minimum amount the user is willing to accept from the swap) to ensure that the user does not get an amount lower than their specified minimum. However, the transfer fees are not taken into account in this comparison. These fees will be deducted from the output amount, and consequently, the user receives less than they expected.

>_ programs/cp-amm/src/instructions/ix_swap.rs RUST

```rust
pub fn handle_swap(ctx: Context<SwapCtx>, params: SwapParameters) -> Result<()> {
    [...]
    require!(
        swap_result.output_amount >= minimum_amount_out,
        PoolError::ExceededSlippage
    );
    [...]
}
```

Remediation

Ensure the check considers the transfer fee by adjusting the `minimum_amount_out` to reflect the expected reduction due to fees.

Code Refactoring OS-MCM-SUG-03

Description

1. Currently, the vesting account is simply initialized without any deterministic address. Convert the `vesting` account into a Program Derived Address (PDA) utilizing the index field from `VestingParameters` as a seed when deriving the address.

>_ programs/cp-amm/src/instructions/ix_lock_position.rs RUST

```rust
pub struct LockPositionCtx<'info> {
    [...]
    #[account(
        init,
        payer = payer,
        space = 8 + Vesting::INIT_SPACE
    )]
    pub vesting: AccountLoader<'info, Vesting>,
    [...]
}
```

2. The `refresh_vesting` instruction currently allows anyone to call it since there are no ownership checks on the owner account. However, when a vesting account is marked as done, it is closed, and its remaining lamports are returned to the owner. In `lock_position`, the payer may not be the owner. However, in `refresh_vesting`, lamports are returned only to the owner. Modify `refresh_vesting` to be owner-only and allow a choice of rent-receiver.
3. In `get_current_base_fee_numerator`, the fee may be reduced over time utilizing a reduction factor. If the base fee is not properly managed when the `reduction_factor` is zero, the system may allow the base fee to increase over time instead of decreasing. Directly return `ONE` if `base == ONE`.

Remediation

Refactor the code as stated above.

Unutilized/Redundant Code

OS-MCM-SUG-04

Description

1. The `fee_claimer` field in `Position` appears redundant, as the verification during `claim_position_fee` is done against `position_nft_account.owner` instead of the `fee_claimer` field.
2. In `fee_parameters::get_min_base_fee_numerator`, `self.number_of_period` is already of type `u16`, and therefore there is no need to perform a conversion (via `try_from`) to `u16`.

>_ *programs/cp-amm/src/params/fee_parameters.rs* RUST

```rust
pub fn get_min_base_fee_numerator(&self) -> Result<u64> {
    let fee_scheduler_mode = FeeSchedulerMode::try_from(self.fee_scheduler_mode)
        .map_err(|_| PoolError::TypeCastFailed)?;
    match fee_scheduler_mode {
        [...]
        FeeSchedulerMode::Exponential => {
            let period =
                u16::try_from(self.number_of_period).map_err(|_|
                    ↪  PoolError::MathOverflow)?;
            let fee_numerator =
            [...]
        }
    }
}
```

3. In `fee_parameters::validate`, the `fee_percent` check is redundant since it is already checked in `pool_fees.validate` that `partner_fee_percent < 100`.

>_ *programs/cp-amm/src/params/fee_parameters.rs* RUST

```rust
pub fn validate(&self) -> Result<()> {
    if self.have_partner() {
        require!(self.fee_percent <= 100, PoolError::InvalidFee);
    } else {
        require!(self.fee_percent == 0, PoolError::InvalidFee);
    }
    validate_fee_fraction(self.fee_percent.into(), 100)
}
```

Remediation

Remove the redundant and unutilized code instances.

Code Clarity

OS-MCM-SUG-05

Description

1. The comment in `fee_math::get_fee_in_period` suggests that by adding 1 to bps, the result should be 1.0001 in the `Q64.64` format. However, this is misleading in the context of calculating `(1-reduction_factor/10_000)`. Also, the comment in `PoolStatus` enumeration is irrelevant.

   >_ *programs/cp-amm/src/math/fee_math.rs* RUST

   ```rust
   pub fn get_fee_in_period(
       cliff_fee_numerator: u64,
       reduction_factor: u64,
       passed_period: u16,
   ) -> Result<u64> {
       [...]
       // Add 1 to bps, we get 1.0001 in Q64.64
       let base = ONE.safe_sub(bps)?;
       let result = pow(base, passed_period.into()).ok_or_else(|| PoolError::MathOverflow)?;
       [...]
   }
   ```

2. There is a typographical error in `PoolFeeParamters` structure name. Ensure to rectify the spelling.
3. Some instructions in the `partner` folder, such as `ix_initialize_reward`, are intended for admin utilization, while others, such as `reward_funder`, are not necessarily limited to partners.
4. `position::get_total_reward` returns only the pending rewards, not the `total_claimed_rewards`, which may be misleading. Renaming it to something like `get_pending_reward` will improve clarity.

Remediation

Implement the above-mentioned suggestions.

A — Uniswap V3 Reference

From the formula (6.15) in Uniswap V3 paper

$$\Delta\frac{1}{\sqrt{P}} = \frac{\Delta x}{L}$$

expand $\Delta\frac{1}{\sqrt{P}}$ as $\frac{1}{\sqrt{P'}} - \frac{1}{\sqrt{P}}$

$$\frac{1}{\sqrt{P'}} - \frac{1}{\sqrt{P}} = \frac{\Delta x}{L}$$

move $\frac{1}{\sqrt{P}}$ to the right side

$$\frac{1}{\sqrt{P'}} = \frac{\Delta x}{L} + \frac{1}{\sqrt{P}}$$

combine the fractions

$$\frac{1}{\sqrt{P'}} = \frac{L + \Delta x \cdot \sqrt{P}}{\sqrt{P} \cdot L}$$

take the reciprocal

$$\sqrt{P'} = \frac{\sqrt{P} \cdot L}{L + \Delta x \cdot \sqrt{P}}$$

Figure A.1: A derivation that leads to the result $\sqrt{P'} = \sqrt{P} * L/(L + x * \sqrt{P})$

B — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

C — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.