OtterSec

MeteoraAg DLMM

Security Assessment

February 24th, 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

01 — Executive Summary

Overview

MeteoraAg engaged OtterSec to assess the `dlmm` program. This assessment was conducted between January 28th and February 18th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 9 findings throughout this audit engagement.

In particular, we identified a vulnerability where it is possible to exploit `go_to_a_bin` by moving the pool out of range and depositing minimal liquidity in intermediate bins, resulting in swaps aborting due to computational costs (OS-MDM-ADV-00). Additionally, excess composition fees are added to liquidity shares due to rounding, allowing for manipulation of share values, resulting in a DoS or loss of funds when users deposit more than the worth of one share (OS-MDM-ADV-01).

Furthermore, it is possible for anyone to call the protocol fee withdrawal function and specify the withdrawal amount, potentially forcing the protocol to incur unnecessary transfer fees when dealing with Token22 assets (OS-MDM-ADV-04).

We also made recommendations to ensure adherence to coding best practices (OS-MDM-SUG-02) and suggested distributing the rewards proportionally based on the liquidity amount in each bin rather than equally per bin (OS-MDM-SUG-00). Additionally, we advised incorporating additional checks within the codebase for improved robustness and security (OS-MDM-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/MeteoraAg/DLMM. This audit was performed against commit 8fbf3f2. We also conducted a follow-up review of PR#353, PR#354, and PR#355.

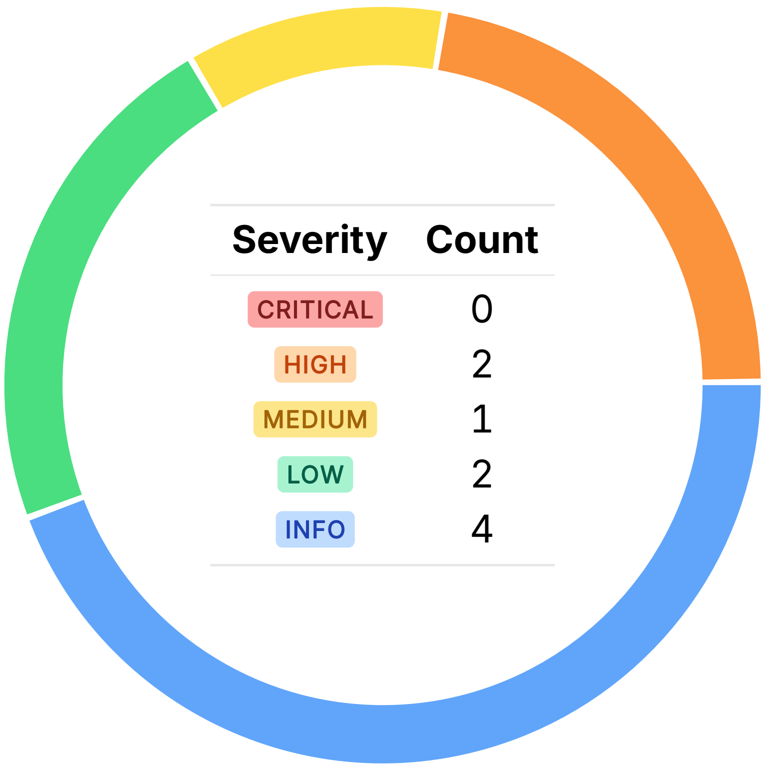
A brief description of the program is as follows:

Name	Description
dlmm	The liquidity book CLMM is a decentralized liquidity management and swapping platform that offers users high flexibility in providing liquidity, earning rewards, and performing swaps. It utilizes a bin-based approach, where liquidity is distributed across different bins, each representing a price range. This allows for concentrated liquidity provisioning, improving capital efficiency for liquidity providers.

03 — Findings

Overall, we reported 9 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	2
MEDIUM	1
LOW	2
INFO	4

04 — 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 A.

ID	Severity	Status	Description
OS-MDM-ADV-00	HIGH	RESOLVED ✓	It is possible to exploit `go_to_a_bin` by moving the pool out of range and depositing minimal liquidity in intermediate bins, resulting in swaps to abort due to computational costs.
OS-MDM-ADV-01	HIGH	RESOLVED ✓	Excess composition fees are added to liquidity shares due to rounding, allowing for manipulation of share values, resulting in a DoS or loss of funds when users deposit more than the worth of one share.
OS-MDM-ADV-02	MEDIUM	RESOLVED ✓	`calculate_pre_fee_amount` improperly derives the `pre_fee_amount`, resulting in an incorrect transfer fee calculation.
OS-MDM-ADV-03	LOW	RESOLVED ✓	Anyone may call `withdraw_protocol_fee` and specify the withdrawal amount, potentially forcing the protocol to incur unnecessary transfer fees when dealing with Token22 assets.
OS-MDM-ADV-04	LOW	RESOLVED ✓	The host fee is deducted separately before the main swap, but the transfer fee is only calculated once on the total amount. This results in the host receiving slightly less than intended due to rounding discrepancies.

Bin Manipulation Resulting in Swap Failure HIGH OS-MDM-ADV-00

Description

It is possible to leverage the `go_to_a_bin` instruction to manipulate the active liquidity bin, disrupting normal swap operations or exploiting rounding mechanics to gain an unfair advantage. Utilizing `go_to_a_bin`, it is possible to change the pool's `lb.active_id` to a bin far from its current position. There is an optimization that skips entirely empty bin arrays, so the attacker must deposit some liquidity to ensure every bin is checked during the swap.

Consequently, when a user attempts to swap tokens, `internal_handle_swap` iterates over the liquidity bins from the `active_id`. Since each intermediate bin now contains some liquidity, the swap process has to traverse through all these bins. This dramatically increases the CU cost due to the need to check each bin, and as a result, the swap will be aborted.

Remediation

Set a minimum liquidity deposit requirement per bin to prevent attackers from utilizing atomic units to populate bins, rendering the attack too expensive to perform.

Patch

Resolved in 43b3a79.

Liquidity Share Manipulation Risk HIGH OS-MDM-ADV-01

Description

`add_liquidity` may result in rounding issues when calculating liquidity shares for users. Excess composition fees created by rounding errors are added to liquidity shares, rendering them more valuable. When liquidity shares are small, this may be exploited by repeatedly calling `remove_liquidity` to withdraw 99.99% of the liquidity. Each withdrawal reduces the liquidity pool, but the remaining shares become exponentially more valuable due to redistributed rounding errors. This may result in a denial-of-service or loss of funds if users deposit more than the value of one liquidity share.

Remediation

Implement a check for the minimum liquidity share, or return the excess rounding back to the user.

Patch

Resolved in 43b3a79.

Failure to Utilize Ceiling Division MEDIUM OS-MDM-ADV-02

Description

`token2022::calculate_pre_fee_amount` should perform a ceiling division operation in the calculation of the `raw_pre_fee_amount`. Currently, it replaces the denominator of the divison calculation with `ONE_IN_BASIS_POINTS`. The original intent of the calculation is to scale the numerator according to the fee basis points, ensuring that the result is rounded up appropriately. However, replacing the denominator with `ONE_IN_BASIS_POINTS` incorrectly scales the numerator by a large factor. As a result, transfer fee calculation becomes incorrect.

>_ lb_clmm/src/utils/token2022.rs RUST

```rust
pub fn calculate_pre_fee_amount(transfer_fee: &TransferFee, post_fee_amount: u64) -> Option<u64>
    ↪  {
    [...]
    else {
        let numerator = (post_fee_amount as u128).checked_mul(ONE_IN_BASIS_POINTS)?;
        let denominator = ONE_IN_BASIS_POINTS.checked_sub(transfer_fee_basis_points)?;
        // let raw_pre_fee_amount = ceil_div(numerator, denominator)?;
        let raw_pre_fee_amount = numerator
            .checked_add(ONE_IN_BASIS_POINTS)?
            .checked_sub(1)?
            .checked_div(denominator)?;

        if raw_pre_fee_amount.checked_sub(post_fee_amount as u128)? >= maximum_fee as u128 {
            post_fee_amount.checked_add(maximum_fee)
        }[...]
    }
}
```

Remediation

Update the calculation to utilize `ceil_div` (`ceil_div((numerator + denominator - 1) / denominator)`).

Patch

Resolved in d58175d.

Possibility of Setting Excessive Transfer Fee LOW OS-MDM-ADV-03

Description

It is possible for anyone to call `withdraw_protocol_fee` and specify the withdrawal amounts on behalf of the fee owner. While the instruction correctly validates that the requested withdrawal amounts do not exceed the accumulated protocol fees, it does not enforce an optimal withdrawal strategy to minimize transfer fees when dealing with Token-2022 (T22) tokens that have built-in transfer fees. Thus, if the pool contains a t22 with transfer fees, this may be utilized to make the protocol pay more in transfer fees than necessary.

Remediation

Only allow the fee owner to call `withdraw_protocol_fee`, preventing external actors from choosing arbitrary amounts.

Patch

Resolved in 8e0f9ce.

Host Fee Underpayment LOW OS-MDM-ADV-04

Description

In `swap2`, when a user initiates a swap, a transfer fee is subtracted from the specified `amount_in` (amount of tokens given by the user for the swap). This subtracted amount is what is actually utilized for the swap. This transfer fee is calculated as if it was a single transfer. The issue arises when a host fee is specified (`host_fee_in`). In this case, `perform_swap` performs two separate transfers. First, the host fee is deducted from the user and sent to the host, and in the second transfer, the remaining amount is transferred from the user to the liquidity pool.

>_ lb_clmm/src/utils/token2022.rs RUST

```rust
fn internal_handle_swap2<'c: 'info, 'info>(
    ctx: &Context<'_, '_, 'c, 'info, Swap2<'info>>,
    amount_in: u64,
    remaining_accounts_info: RemainingAccountsInfo,
) -> Result<InternalSwapResult> {
    [...]
    let transfer_fee_excluded_amount_in =
        transfer_fee_excluded_amount_in.safe_sub(total_host_fee)?;

    // very edge case that calculate_transfer_fee_included_amount return result greater than
        ↪ amount_in
    // due to function calculate_inverse_fee is round up
    let transfer_fee_included_amount_in =
        calculate_transfer_fee_included_amount(token_mint_in, transfer_fee_excluded_amount_in)?
            .amount
            .min(amount_in);

    let transfer_fee_included_host_fee = amount_in.safe_sub(transfer_fee_included_amount_in)?;
    [...]
```

However, the calculation in `calculate_transfer_fee_excluded_amount` assumes that only one transfer occurs, so the entire `amount_in` is adjusted for fees before transferring. However, in reality, the host fee is deducted in a separate transaction. Since each transfer is subject to a fee, the host actually receives less than the `total_host_fee` they were supposed to get.

Remediation

Modify `calculate_transfer_fee_excluded_amount` to account for the fact that the host fee is deducted separately. Instead of assuming a single transfer, explicitly compute the transfer fee twice—once for the host fee and once for the swap transfer. This ensures that the host receives the full intended amount after accounting for the transfer fee.

Patch

Resolved in d58175d.

05 — 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-MDM-SUG-00	Rewards are distributed equally across bins crossed, allowing minimal liquidity to disproportionately claim rewards.
OS-MDM-SUG-01	There are several instances where proper validation should be performed to prevent potential security issues.
OS-MDM-SUG-02	Suggestions regarding inconsistencies in the codebase and ensuring adherence to coding best practices.
OS-MDM-SUG-03	There are multiple instances of redundant and un-utilized code that should be removed for better maintainability and clarity.

Unfair Reward Allocation

OS-MDM-SUG-00

Description

Rewards are distributed equally across all bins a position spans, regardless of the liquidity size in each bin. This approach introduces a potential vulnerability, particularly when extremely small (or dust) liquidity deposits are involved. It enables a bin containing just one atomic unit of liquidity to receive the same reward as a bin with a much larger liquidity amount. Thus, a bin with a minimal amount of liquidity (dust) will earn disproportionately large rewards.

Remediation

Distribute rewards proportionally based on the liquidity amount in each bin rather than equally per bin.

Improving Validation Logic

OS-MDM-SUG-01

Description

1. Setting a reasonable upper limit on `increase_oracle_length` is recommended to prevent performance issues created by unlimited reallocation, which may unnecessarily increase account sizes.
2. Utilize checked casts in `add_liquidity_by_strategy`, specifically for calculations in `to_weight_descending_order` and `to_weight_curve`, in order to improve safety and prevent potential overflow or underflow issues.
3. In `handle_exact_out2`, the check that verifies if `exact_out_amount` is equal to `transfer_fee_excluded_amount_out` is necessary to prevent premature loop termination when processing swaps across multiple bins in `internal_handle_swap2`. Without this check, the function may exit early when `amount_left` is zero, without guaranteeing that the required `exact_amount_out` is provided.

   >_ *lb_clmm/src/instructions/v2/swap2.rs* RUST

   ```rust
   pub fn handle_exact_out2<'c: 'info, 'info>([...]) -> Result<()> {
       [...]
       // TODO, do we need this check?
       require!(
           exact_out_amount == transfer_fee_excluded_amount_out,
           LBError::NotExactAmountOut
       );
       Ok(())
   }
   ```

Remediation

Incorporate the above-stated validations.

Code Maturity

OS-MDM-SUG-02

Description

1. There is a minor inconsistency in how bin IDs are checked in `advance_active_bin` and `InitializeBinArray`, as they utilize global constants (`MIN_BIN_ID`, `MAX_BIN_ID`) to validate bin IDs, rather than utilizing the pair-specific fields (`min_bin_id` and `max_bin_id`). It is always more appropriate to check against the pair's fields.
2. Inc `token2022`, the check for `transfer_fee_basis_points == 100%` in `calculate_transfer_fee_included_amount` is redundant because `calculate_pre_fee_amount` will handle this case correctly later in the process.

>_ lb_clmm/src/utils/token2022.rs RUST

```rust
pub fn calculate_transfer_fee_included_amount<'info>(
    token_mint: &InterfaceAccount<'info, Mint>,
    transfer_fee_excluded_amount: u64,
) -> Result<TransferFeeIncludedAmount> {
    [...]
        if u16::from(epoch_transfer_fee.transfer_fee_basis_points) == MAX_FEE_BASIS_POINTS
            ↪ {
            // edge-case: if transfer fee rate is 100%, current SPL implementation returns
                ↪ 0 as inverse fee.
            // https://github.com/solana-labs/solana-program-library/blob/
            // fe1ac9a2c4e5d85962b78c3fc6aaf028461e9026/token/program-2022/src/
            // extension/transfer_fee/mod.rs#L95
            // But even if transfer fee is 100%, we can use maximum_fee as transfer fee.
            // if transfer_fee_excluded_amount + maximum_fee > u64 max, the
                ↪ followingchecked_add should fail.
            u64::from(epoch_transfer_fee.maximum_fee)
        }
    [...]
}
```

3. Validate `bin_step > 0` when initializing `PresetParameter` and `PermissionLbPair` to ensure a valid state and consistency with `CustomizablePermissionlessLbPair`.
4. It is recommended to make the memo program optional in `ModifyLiquidity2` while retaining it, rather than duplicating additional contexts.

Remediation

Implement the above-mentioned suggestions.

Redundant/Unutilized Code

OS-MDM-SUG-03

Description

1. The check to ensure `amount_left == 0` in `handle_exact_in2` (shown below) and `handle_exact_in_with_price_impact2` is redundant because these functions perform exact-in swaps, which always consume the full `amount_in`. The loop inside `internal_handle_swap2` only exits early if an exact-out condition is set, which is not the case here.

>_ lb_clmm/src/instructions/v2/swap2.rs RUST

```rust
pub fn handle_exact_in2<'c: 'info, 'info>(
    ctx: Context<'_, '_, 'c, 'info, Swap2<'info>>,
    amount_in: u64,
    min_amount_out: u64,
    remaining_account_info: RemainingAccountsInfo,
) -> Result<()> {
    [...]
    // Since swap is swapExactsTokenForToken, therefore amount_in must be swap finish
    require!(amount_left == 0, LBError::PairInsufficientLiquidity);
    [...]
    Ok(())
}
```

2. There is significant duplicate logic across the different `add_liquidity2` functions. It is recommended that this logic be consolidated for efficiency.
3. The `CompressedBinDepositAmount2` structure in `add_liquidity_single_side_precise2` is unnecessary and never utilized, and should be removed.

>_ lb_clmm/src/instructions/v2/add_liquidity_single_side_precise2.rs RUST

```rust
#[derive(AnchorSerialize, AnchorDeserialize, Debug, Clone)]
pub struct CompressedBinDepositAmount2 {
    pub bin_id: i32,
    pub amount: u32,
}
```

Remediation

Remove the redundant and un-utilized code instances listed above.

A — 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.

B — 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.