Meteora

Security Assessment

February 28th, 2024 — Prepared by OtterSec

Thibault Marboud thibault@osec.io

Ajay Shankar Kunapareddy d1r3wolf@osec.io

Table of Contents

01 — Executive Summary

Overview

Meteora engaged OtterSec to assess the `lb-clmm` program. This assessment was conducted between January 18th and February 8th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we discovered a vulnerability concerning ambiguity in determining the swap direction when the start and end indices are equal (OS-MRA-ADV-00) and a potential denial of service during withdrawal when the user's liquidity share is a small fraction of the total pool liquidity (OS-MRA-ADV-02). Additionally, we highlighted the absence of verification for the actual ownership of token accounts during withdrawal (OS-MRA-ADV-01).

We also provided suggestions regarding the incorporation of additional validations and checks (OS-MRA-SUG-00) and enhancing the overall quality of the code base (OS-MRA-SUG-01). Furthermore, we proposed simplifying the logic to improve readability and eliminate redundancy in the code (**??**).

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 907b80c.

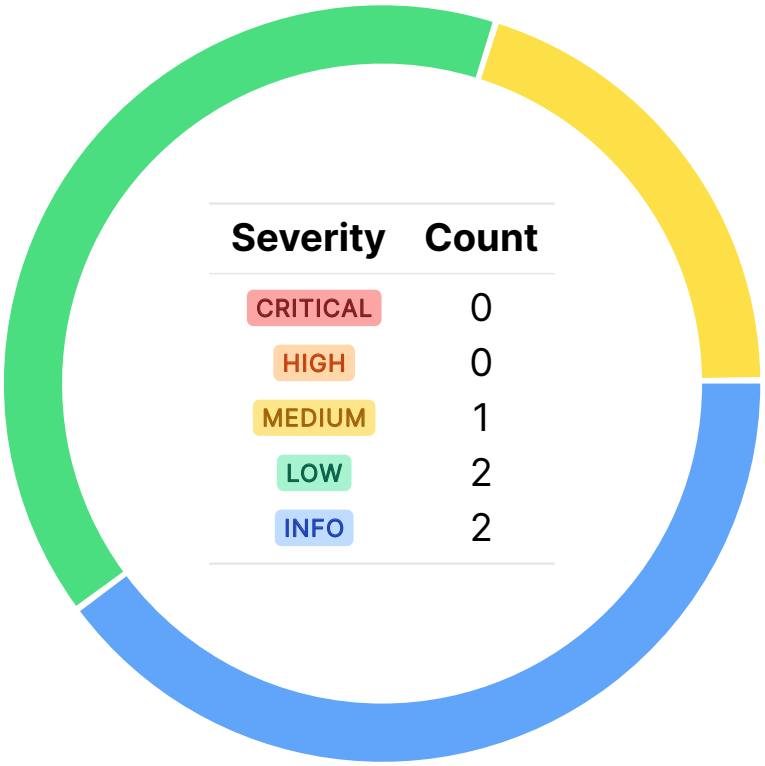
A brief description of the programs is as follows:

Name	Description
lb-clmm	A Solana dynamic liquidity market maker that implements the Liquidity Book protocol, conceptualized by Trader Joe, with capabilities for on-hain farming and manual claiming of pool fees.

03 — Findings

Overall, we reported 5 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	0
MEDIUM	1
LOW	2
INFO	2

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-MRA-ADV-00	MEDIUM	RESOLVED	Ambiguity in determining the swap direction when `next_bin_array_index_with_liquidity` encounters equal start and end indices.
OS-MRA-ADV-01	LOW	RESOLVED	`validate_outflow_to_ata_of_position_owner` lacks any verification for the actual ownership of token accounts during withdrawal.
OS-MRA-ADV-02	LOW	RESOLVED	The withdrawal functions may result in a denial of service vulnerability if the user liquidity share is a small fraction of the total.

Ambiguity In Swap Direction MEDIUM OS-MRA-ADV-00

Description

`next_bin_array_index_with_liquidity`, in `BinArrayBitmapExtension`, determines the direction of the search (up or down) based on whether the start index is greater or less than the end index. When the start and end indices are equal, there is ambiguity in the value of `swap_for_y` and consequently in determining the direction of the search. When the start and end indices are equal, the function does not have a clear indication of whether it should search up or down. This may yield unexpected behavior in the search direction.

Proof of Concept

>_ *lb_clmm/src/state/bin_array_bitmap_extension.rs* rust

```rust
#[test]
let mut extension = BinArrayBitmapExtension::default();
extension.flip_bin_array_bit(600).unwrap();

let (index, flag) = extension.next_bin_array_index_with_liquidity(false, 550).unwrap();
println!("Swap for X (Index: 550): {} {}", index, flag); // Output: 600 since index increases
    ↪ from 550
let (index, flag) = extension.next_bin_array_index_with_liquidity(true, 600).unwrap();
println!("Swap for Y (Index: 650): {} {}", index, flag); // Output: 600 since index decreases
    ↪ from 650
let (index, flag) = extension.next_bin_array_index_with_liquidity(true, 513).unwrap();
println!("Swap for Y (Index: 513): {} {}", index, flag); // Output: 511 since index decreases
    ↪ from 513

// Vulnerable Case, Swapping for Y: searching index from 512
let (index, flag) = extension.next_bin_array_index_with_liquidity(true, 512).unwrap();
println!("Swap for Y (Index: 512): {} {}", index, flag);
```

In the above test case, where the start and end indices are both 512, the function should search in the decreasing order of indices, as indicated by the `swap_for_y` parameter equaling true. However, due to the ambiguity in handling equal start and end indices, the function incorrectly assumes that it should search in the increasing order of indices (from 512 to 600).

Remediation

Ensure `next_bin_array_index_with_liquidity` explicitly handles the case where the start and end indices are equal. It should either choose a default direction (up or down) or return an error indicating the ambiguity.

Patch

Fixed in 37c1593 by handling the case where the start and end indices are equal, separately.

Missing Ownership Verification In Token Withdrawal LOW OS-MRA-ADV-01

Description

`validate_outflow_to_ata_of_position_owner` fails to check whether the owner of the associated token accounts (`self.user_token_x` and `self.user_token_y`) matches the expected owner of the position. The function utilizes the associated token addresses (`dest_token_x` and `dest_token_y`) to check whether the provided associated token accounts (`self.user_token_x` and `self.user_token_y`) are the expected destination for the withdrawal.

>_ *lb_clmm/src/instructions/claim_fee.rs* rust

```rust
impl<'a, 'b, 'c, 'info> PositionLiquidityFlowValidator for ClaimFee<'info> {
    fn validate_outflow_to_ata_of_position_owner(&self, owner: Pubkey) -> Result<()> {
        let dest_token_x = anchor_spl::associated_token::get_associated_token_address(
            &owner,
            &self.token_x_mint.key(),
        );
        require!(
            dest_token_x == self.user_token_x.key(),
            LBError::WithdrawToWrongTokenAccount
        );

        let dest_token_y = anchor_spl::associated_token::get_associated_token_address(
            &owner,
            &self.token_y_mint.key(),
        );
        require!(
            dest_token_y == self.user_token_y.key(),
            LBError::WithdrawToWrongTokenAccount
        );
        Ok(())
    }
}
```

However, the function does not verify that the actual owner of the associated token accounts (`self.user_token_x` and `self.user_token_y`) matches the expected owner of the position. It assumes that the correct entity owns the associated token accounts based on the associated token addresses.

Remediation

Perform an additional check to ensure that the owner field of the associated token accounts (`self.user_token_x` and `self.user_token_y`) matches the expected owner (`owner`) before allowing the withdrawal.

Patch

Fixed in 92e32d0 by adding owner validation for the associated token accounts.

Denial Of Service In Withdrawal LOW OS-MRA-ADV-02

Description

Within `remove_liquidity` and `remove_all_liquidity`, while attempting to remove liquidity, if a users liquidity is a tiny fraction of the total liquidity in the pool (e.g., 1 out of 1000 shares), and the withdrawal calculations result in negligible amounts for both `amount_x` and `amount_y` (example: `amount_x` and `amount_y` are both at 100), the withdrawal calculations would round down to zero ($(1 * 100/1000) = 0$) for both `x` and `y`.

Consequently, if both `amount_x` and `amount_y` are zero, the condition: `amount_x.safe_add(amount_y)? > 0`, is not satisfied, resulting in a withdrawal rejection. As a result, the user is unable to withdraw their funds even though they have a legitimate share of liquidity, creating a situation where the user's funds are effectively locked in the liquidity pool, impacting their ability to manage their assets.

Remediation

Implement a minimum withdrawal threshold to allow users to withdraw even if the calculated amounts are minimal.

Patch

Fixed in 93408c4.

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-MRA-SUG-00	Incorporating missing validations and checks into the code base.
OS-MRA-SUG-01	Recommendations for improving the quality of the code base.

Code Integrity OS-MRA-SUG-00

Description

1. To prevent unexpected collisions, we recommended prefixing the seed of PDA accounts with a unique constant for each account type.
2. Upon initialization of a new `bin_array`, the program could ensure that the `lower_bin_id` and `upper_bin_id` fall within the bounds set by the `min_bin_id` and `max_bin_id` of the `lb_pair.parameters`.
3. The `removeLiquidity` and `removeAllLiquidity` instructions could be enhanced by incorporating a slippage check to safeguard users from unexpected liquidity movements. This can be achieved by introducing two parameters, `amountXMin` and `amountYMin`, which represent the minimum amounts of tokens users anticipate receiving.
4. Ensure coherence between the designated bin IDs in the `LiquidityParameterByWeight` structure and the actual bin IDs range of the position (`position.lower_bin_id` and `position.upper_bin_id`). If `first_bin_id` is less than `position.lower_bin_id` or `last_bin_id` is greater than `position.upper_bin_id`, it may lead users to attempt depositing liquidity into bins that are not part of the position.

>_ *lb_clmm/src/instructions/add_liquidity_by_weight.rs* rust

```rust
fn validate<'a, 'info>(&'a self, active_id: i32) -> Result<()> {
        let bin_count = self.bin_count();
        require!(bin_count > 0, LBError::InvalidInput);
        require!(
            bin_count <= MAX_BIN_PER_POSITION as u32,
            LBError::InvalidInput
        );
        let bin_shift = if active_id > self.active_id {
            active_id - self.active_id
        } else {
            self.active_id - active_id
        };
        require!(
            bin_shift <= self.max_active_bin_slippage.into(),
            LBError::ExceededBinSlippageTolerance
        );
        [...]
}
```

Code Quality & Maturity OS-MRA-SUG-01

Description

The code quality is good, reflecting the efforts invested in the project. However, there are multiple unnecessary return statements, unnecessary references, and several instances where the code is converting between types unnecessarily.

Some other recommendations that could also be improved:

- *get_seconds_elapsed_since_last_update* function could use *calculate_reward_accumulated_since_last_update* to compute *time_period* instead of duplicating code.
- *BinArray::get_bin_index_in_array* could be simplified by using *bin_id.safe_sub(lower_bin_id)* to handle negative *bin_id* scenarios, avoiding the calculation involving *upper_bin_id*.

>_ *lb_clmm/src/state/bin.rs* rust

```rust
(MAX_BIN_PER_ARRAY as i32).safe_sub(upper_bin_id.safe_sub(bin_id)?)?.safe_sub(1)?
```

- *out_amounts* variable is unused in *remove_liquidity* instruction.

>_ *lb_clmm/src/instructions/remove_liquidity.rs* rust

```rust
pub fn handle<'a, 'b, 'c, 'info>(
    ctx: Context<'a, 'b, 'c, 'info, ModifyLiquidity<'info>>,
    bin_liquidity_reduction: Vec<BinLiquidityReduction>,
) -> Result<()> {
    [...]
    let mut out_amounts = vec![[0u64; 2]; bin_liquidity_reduction.len()];
    [...]
}
```

Remediation

In an effort to elevate the code quality even further and introduce consistency across the entire project (or even multiple projects), it could be beneficial to use a linter tool such as Clippy. Since issues tend to accumulate, the effort required may be higher when introducing such a tool to an existing project. However, the consistency it provides can be beneficial when collaborating with different individuals across various projects.

Note: When using Clippy with Anchor, it is recommended to disable warnings about large error types: `#![allow(clippy::result_large_err)]`.

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 the other 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.