Zellic

October 27, 2023

Prepared for
Boris Povod
Igor Demko
Pontem Network

Prepared by
Aaron Esau
Daniel Lu
Yuhang Wu
Zellic

Liquidswap
Smart Contract Security Assessment

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue ↗, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io ↗ or follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io ↗.

1. Executive Summary

Zellic conducted a security assessment for Pontem Network from October 16th to 27th, 2023. During this engagement, Zellic reviewed Liquidswap's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an on-chain attacker exploit the flash-loan feature to drain the liquidity pools?
- Is there any potential for a deadlock in the system, especially concerning the concentrated liquidity feature?
- Could a malicious message or input from a user lead to a lockup or drainage of user funds?
- Could current flash-loan design allow users to bypass or avoid fees?
- Is there any potential issue with the design of the bins and pools?

1.2. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

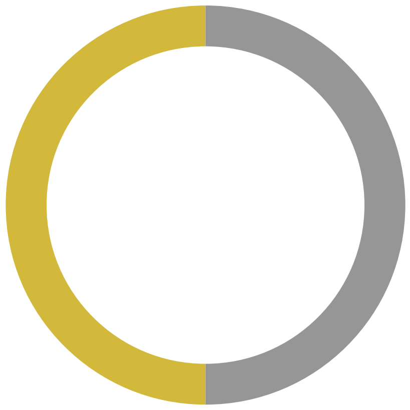
1.3. Results

During our assessment on the scoped Liquidswap modules, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Pontem Network's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	1
🟩 Low	0
⬜ Informational	1



2. Introduction

2.1. About Liquidswap

Liquidswap is a concentrated liquidity AMM DEX, drawing substantial reference from Trader Joe V2.1. It is written in Move and is slightly more configurable than the original Trader Joe.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the modules.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an “Informational” finding higher than a “Low” finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients’ threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Liquidswap Modules

Repository	https://github.com/pontem-network/liquidswap_v1 ↗
Version	liquidswap_v1: `981fab31bf5bb134f21b66ad533f127a6c96bc5f`
Programs	• bin_steps/sources/bin_steps.move • math_helpers/sources/bit_math.move • math_helpers/sources/fp128_math.move • math_helpers/sources/fp64_math.move • math_helpers/sources/tree_math.move • sources/config.move • sources/emergency.move • sources/lb_token.move • sources/libs/bin_helper.move • sources/libs/fees_helper.move • sources/oracle.move • sources/pool.move • sources/treasury.move
Type	Move
Platform	Aptos

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of three person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Aaron Esau
Engineer
aaron@zellic.io ↗

Daniel Lu
Engineer
daniel@zellic.io ↗

Yuhang Wu
Engineer
yuhang@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 16, 2023	Kick-off call
October 16, 2023	Start of primary review period
October 27, 2023	End of primary review period

3. Detailed Findings

3.1. Flash-loan design lets users avoid fees

Target	**liquidswap_v1::pool**		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The protocol supports flash loans where users can borrow funds from both sides of any pool, as long as they are returned by the end of the transaction with a fee. In the `pool::flashloan` function, the funds are extracted from the pool and given to the user, along with a "receipt" of the transaction.

```
let loan_coins_x = coin::extract(&mut pool.coins_x, amount_x);
let loan_coins_y = coin::extract(&mut pool.coins_y, amount_y);

let flashloan = Flashloan<X, Y, BinStep> {
    loan_x: amount_x,
    loan_y: amount_y,
};

(loan_coins_x, loan_coins_y, flashloan)
```

Then, while the funds and receipt are returned, the protocol ensures that the required fees are paid. These fees are distributed to the liquidity providers in the active price bin.

```
// Actual protocol fee from liquidity provider fee.
// Everything that on top of (fee - protocol fee) goes to treasury.
let (pfee_x, pfee_y) = fees_helper::get_protocol_fee_amount_x_y(
    &pool.fee_params,
    lp_fee_x, lp_fee_y,
);

// Add fees into active bin.
let bin_fee_x = lp_fee_x - pfee_x;
let bin_fee_y = lp_fee_y - pfee_y;
let bin_fee_x_coins = coin::extract(&mut coin_x_loan, bin_fee_x);
let bin_fee_y_coins = coin::extract(&mut coin_y_loan, bin_fee_y);

active_bin.reserves_x = active_bin.reserves_x + bin_fee_x;
active_bin.reserves_y = active_bin.reserves_y + bin_fee_y;
```

```
coin::merge(&mut pool.coins_x, bin_fee_x_coins);
coin::merge(&mut pool.coins_y, bin_fee_y_coins);

// The rest going to treasury as protocol fee.
protocol_fee_x = protocol_fee_x - bin_fee_x;
protocol_fee_y = protocol_fee_y - bin_fee_y;
```

To summarize, while users can take flash loans and utilize the liquidity of the entire pool, the rewards are returned only to liquidity providers in the active bin (which have much more shallow liquidity).

Impact

During a flash loan, this fee structure allows some rewards to be stolen, either by front-running or by the user themselves. Suppose that a pool has 10 bins, each with 10 USDC and 10 USDT. We will assume that the pool has a 20% fee for flash loans.

Suppose that we want to borrow all 100 USDC in the pool. Normally, this would cost us 20 additional USDC in fees at the end of the transaction. Instead, we can first take a 20 USDC flash loan from a different AMM — let us imagine that its flash loan fee is 20% as well. If we deposit these tokens into the active bin, we will receive around half of the active bin's shares.

Now, let us take the 100 USDC loan we want from this pool. After we utilize the funds in the transaction, we return them along with the required 20% fee. Then, this 20 USDC fee is distributed among only the active bin, but half of the bin is under our control. So, if we immediately burn our share, we exit the pool with 30 tokens: 20 from our original deposit and 10 from our flash loan's fee.

Finally, if we return the 20 tokens for the original flash loan along with the four-token fee to the other AMM, we are left with six extra tokens. This lets us save a significant amount of fees on our flash loan.

Recommendations

We recommend that Pontem Network distributes fees to all liquidity providers, instead of just those in the active bin. Conceptually, this means that liquidity providers would be paid exactly according to their contribution to the flash loan — making these types of attacks impossible. Practically, this could be done by adding global stake accounting, since iterating over all bins would be too expensive.

Remediation

This issue has been acknowledged by Pontem Network.

3.2. Unexpected overflow in high and low bins

Target	liquidswap_v1::pool		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

Since pools may contain bins with very high and very low prices, there are a number of overflow conditions they must manage. The first relates to computing the price for a given bin; the second relates to computing bin liquidity based on price and balance.

For instance, the pool module allows liquidity to be minted in bins with IDs ranging from $[0, 2^{24})$. However, prices can only be computed when bin IDs lie in $(2^{23} - 2^{20}, 2^{23} + 2^{20})$. This is due to the following check in the liquidswap_v1::fp128_math module.

```
fun unsigned_power_fp128(base_fp128: u256, y: u32): u256 {
    // (y == 0) case is handled in the upper level.
    assert!(y > 0, ERR_ZERO_EXPONENT);
    assert!(y < 0x100000, ERR_EXPONENT_OVERFLOW);
```

So, bin mechanics — which fortunately include bin creation — are broken when their IDs lie outside this interval, despite them being seemingly permitted by the pool.

Additionally, there are many more situations where computing a bin's *price* does not overflow, but arithmetic with the left liquidity does. For example, a bin ID of 443637 (or $2^{23} + 443637$ unsigned) is enough to break the liquidity computation for any bin step and any left balance.

These conditions are not handled explicitly by the pool; rather, they result in reverts caused either by assertions in the math libraries or by arithmetic on the Move VM.

Impact

This causes some pool operations to fail with unclear error messages. More importantly, these overflow conditions could make the protocol difficult to maintain. Although these failures would likely never result in stolen funds, there may be risk of broken pool mechanics or locked liquidity. Future maintenance of the protocol needs to be performed with these edge cases in mind, which is more difficult when they are handled *implicitly*.

Recommendations

We recommend handling these edge cases in the pool logic itself.

Remediation

This issue has been acknowledged by Pontem Network.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Slippage protection during minting

The router module provides a public `add_liquidity` function for minting liquidity in a given pool. When interacting with this API, the user provides a list of bins and the amount to deposit in each. Because it is possible for a pool's active bin to change while the transaction is pending, this function will offset the input bin IDs (within limits provided by the user).

```
let i = 0;
let len = vector::length(&bin_ids);
if (active_bin_id > active_bin_id_desired) {
    let bin_shift = active_bin_id - active_bin_id_desired;

    // Shift right.
    while (i < len) {
        let bin_id = vector::borrow_mut(&mut bin_ids, i);

        *bin_id = *bin_id + bin_shift;
        assert!(*bin_id < MAX_U24,
    ERR_INVALID_BIN_ID_AFTER_LIQUIDITY_SHIFT);

        i = i + 1;
    }
} else {
    let bin_shift = active_bin_id_desired - active_bin_id;

    // Shift left.
    while (i < len) {
        let bin_id = vector::borrow_mut(&mut bin_ids, i);

        assert!(*bin_id >= bin_shift,
    ERR_INVALID_BIN_ID_AFTER_LIQUIDITY_SHIFT);
        *bin_id = *bin_id - bin_shift;

        i = i + 1;
    }
};
```

Additionally, this function takes parameters `amount_x_min` and `amount_y_min`, which let the user limit how many tokens can be left over after the deposit. However, this amount is calculated based on the total number of tokens spent, and does not distinguish between funds used to mint liquidity and funds used to pay fees.

This could be important because the minting process incurs a number of fees. First, the protocol takes some constant percentage of the deposited funds. Second, liquidity providers may be charged *composition fees*, which are a mechanism used to prevent users minting and burning liquidity for the purpose of bypassing fees while performing an exchange. The `add_liquidity` API does not allow depositors to limit such fees, which could discourage minting in high-volatility scenarios.

4.2. Test coverage

Overall, the project has high code quality, and includes an extensive test suite with code coverage of over 98%. These tests include both unit tests and integration testes, and cover both positive and negative behavior. We encourage Pontem Network to include further tests for more realistic usage scenarios, such as when there is more than one account interacting with the protocol.

It is important to note that high code coverage is not a security guarantee: the state space of any project is almost certainly much larger than the number of lines of code. We notice that some modules, such as `liquidswap_v1::config`, already include specifications for formal verification by the Move prover. We recommend futher taking advantage of Move's formal verification tools, and adopting additional fuzz testing as well.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the modules and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: tree_math.move

Function: `find_first_bin()`

Given a bin tree and a bin, this finds the ID of the closest existing bin.

Inputs

- `tree`
- `bin_id`
    - **Precondition**: Must be within valid `bin_id` range (fits in `u24`).
- `right`

Branches and code coverage (including function calls)

Intended branches

- ☑ Finds closest bin on left and right side.
- ☑ Correctly finds closest bin when top 16 bits are the same.
- ☑ Correctly finds closest bin when top eight bits are the same.
- ☑ Correctly finds closest bin when top eight bits differ.
- ☑ Returns none when the chosen ID is last in the given direction.

Function: `remove_from_tree()`

Given a mutable bin tree and a bin ID, this removes the ID from the tree.

Inputs

- `tree`
- `bin_id`
    - **Precondition**: Must be within valid `bin_id` range (fits in `u24`).

Branches and code coverage (including function calls)

Intended branches

- ☑ Removes a valid `bin_id` from the tree.
- ☑ Correctly updates upper layers when a layer becomes zero.
- ☐ Does not modify tree if `bin_id` already does not exist.

Function: `add_to_tree()`

Given a mutable bin tree and a bin ID, this adds the ID from the tree.

Inputs

- `tree`
- `bin_id`
    - **Precondition**: Must be within valid `bin_id` range (fits in `u24`).

Branches and code coverage (including function calls)

Intended branches

- ☑ Adds a valid `bin_id` to the tree.
- ☐ Does not modify tree if `bin_id` already exists.

5.2. Module: router.move

Function: `swap_exact_x_for_y()`

This swaps an exact amount of X coin for a minimum amount of Y coin.

Inputs

- `x_coins_in`
    - **Validation**: Must be a valid Coin<X> object with a value > 0.
    - **Impact**: Controls the amount of X coin being swapped.
- `y_coins_out_min_val`
    - **Validation**: Must be `u64` integer type with a value > 0.
    - **Impact**: Sets a lower bound on the amount of Y coin received.

Branches and code coverage (including function calls)

Intended branches

- ☑ `swap_exact_x_for_y` executes successfully.

Negative behaviour

- ☑ Error if output is less than minimum.

Function call analysis

- `swap_exact_x_for_y() -> pool::swap_x_for_y(x_coins_in)`
  - **External/Internal**: Internal .
  - **Argument control**: User controls `x_coins_in` amount.
  - **Impact**: Controls amount of Y coin received up to pool math limits.

Function: `swap_exact_y_for_x()`

Swaps an exact amount of Y coin for a minimum amount of X coin.

Inputs

- `y_coins_in`
  - **Validation**: Must be a valid Coin<Y> object with a value > 0.
  - **Impact**: Controls the amount of Y coin being swapped.
- `x_coins_out_min_val`
  - **Validation**: Must be `u64` integer type with a value > 0.
  - **Impact**: Sets a lower bound on the amount of X coin received.

Branches and code coverage (including function calls)

Intended branches

- ☑ `swap_exact_y_for_x` executes successfully.

Negative behaviour

- ☑ Error if output is less than minimum.

Function call analysis

- `swap_exact_y_for_x() -> pool::swap_y_for_x(y_coins_in)`
  - **External/Internal**: Internal.

  - **Argument control**: User controls `y_coins_in` amount.
  - **Impact**: Controls amount of X coin received up to pool math limits.

Function: `swap_x_for_exact_y()`

Swaps X coin for an exact amount of Y coin.

Inputs

- `x_coins_in`
  - **Validation**: Must be a valid Coin<X> object with a value > 0.
  - **Impact**: Sets max amount of X that can be swapped.
- `y_coins_required_out`
  - **Validation**: Must be `u64` integer type with a value > 0.
  - **Impact**: Sets the exact output amount of Coin Y.

Branches and code coverage (including function calls)

Intended branches

- ☑ `swap_x_for_exact_y` executes successfully.

Negative behaviour

- ☑ Error if insufficient output coins.

Function call analysis

- `swap_x_for_exact_y() -> pool::get_amount_in(y_coins_required_out, false)`
  - **External/Internal**: Internal.
  - **Argument control**: User controls `y_coins_required_out`.
  - **Impact**: Calculates required input amount.
- `swap_x_for_exact_y() -> pool::swap_x_for_y(x_coins_in)`
  - **External/Internal**: Internal.
  - **Argument control**: `x_coins_in` restricted by `get_amount_in()`.
  - **Impact**: Swaps coins up to pool math limits.

Function: `swap_y_for_exact_x()`

Swaps Y coin for an exact amount of X coin.

Inputs

- `y_coins_in`
  - **Validation**: Must be a valid Coin<Y> object with a value > 0.
  - **Impact**: Sets max amount of Y coin that can be swapped.
- `x_coins_required_out`
  - **Validation**: Must be `u64` integer type with a value > 0.
  - **Impact**: Sets the exact output amount of X coin.

Branches and code coverage (including function calls)

Intended branches

- ☑ `swap_y_for_exact_x` executes successfully.

Negative behaviour

- ☑ Error if insufficient output coins.

Function call analysis

- `swap_y_for_exact_x() -> pool::get_amount_in(x_coins_required_out, true)`
  - **External/Internal**: Internal.
  - **Argument control**: User controls `x_coins_required_out`.
  - **Impact**: Calculates required input amount.
- `swap_x_for_exact_y() -> pool::swap_y_for_x(y_coins_in)`
  - **External/Internal**: Internal.
  - **Argument control**: `y_coins_in` restricted by `get_amount_in()`.
  - **Impact**: Swaps coins up to pool math limits.

5.3. Module: lb_token.move

Function: `mint_token(minter, token_data_id, amount)`

This mints a new token with the given token data ID and amount.

Inputs

- `minter`
  - **Validation**: Checks that the minter is a signer who can mint tokens.
  - **Impact**: Ensures only allowed accounts can mint tokens.
- `token_data_id`

    - **Validation**: No validation.
    - **Impact**: Specifies unique ID of token data to mint, allows minting specific tokens in a collection.
- `amount`
    - **Validation**: No validation.
    - **Impact**: Specifies number of tokens to mint, allows minting desired amount.

Branches and code coverage (including function calls)

Intended branches

- ☑ Checks that token was minted with correct amount.

Negative behavior

- ☑ Checks for invalid minting results.

Function call analysis

- `mint_token -> token::mint_token(minter, token_data_id, amount)`
    - **External/Internal**: External module call.
    - **Argument control**:
        - `minter` controlled by function argument.
        - `token_data_id` controlled by function argument.
        - `amount` controlled by function argument.
    - **Impact**: Mints amount of tokens with `token_data_id` to minter.
- `mint_token -> token::withdraw_token(minter, token_id, amount)`
    - **External/Internal**: External module call.
    - **Argument control**:
        - `minter` controlled by function argument.
        - `token_id` returned by previous mint call.
        - `amount` controlled by function argument.
    - **Impact**: Withdraws newly minted tokens to minter.

5.4. Module: lb_token.move

Function: `increase_oracle_length<X, Y, BinStep>(samples_to_add: u64)`

This increases the length of the oracle by `samples_to_add`.

Inputs

- `samples_to_add`
    - **Validation**: Assert `samples_to_add > 0`.
    - **Impact**: Controls how many samples are added to the oracle. Can increase storage cost.

Branches and code coverage

Intended branches

- ☑ Check `increase_oracle_length` executes successfully.

Negative behaviour

- ☑ Revert with `ERR_ZERO_SAMPLES_TO_ADD` if `samples_to_add <= 0`.
- ☑ Revert with `ERR_MAX_LENGTH_REACHED` if new length > `MAX_ORACLE_LENGTH`.

Function: `update_oracle<X, Y, BinStep>(active_bin_id: u32, volatility_accum: u32, bins_crossed: u32)`

This updates the active sample in the oracle with new data.

Inputs

- `active_bin_id`
    - **Validation**: None.
    - **Impact**: Provides active bin ID to calculate `cum_active_id`.
- `volatility_accum`
    - **Validation**: None.
    - **Impact**: Provides volatility data to calculate `cum_vol_accum`.
- `bins_crossed`
    - **Validation**: None.
    - **Impact**: Provides bins crossed data to calculate `cum_bins_crossed`.

Branches and code coverage

Intended branches

- ☑ Check `update_oracle` executes successfully in all cases.

Negative behavior

- ☑ Check for cases when lifetime is exceeded.

☑ Check for cases when called twice in the same second.

Function Call Analysis

- `update_oracle -> now_seconds()`
    - **External/Internal**: External system module.
    - **Argument control**: No.
    - **Impact**: Provides current timestamp to calculate lifetime.

5.5. Module: lb_token.move

Function: `burn<X, Y, BinStep>(liq_nfts: vector<Token>)`

This burns LB tokens to withdraw deposited X and Y liquidity.

Inputs

- `liq_nfts`
    - **Validation**: The length should be larger than zero.
    - **Impact**: LB tokens to burn.

Branches and code coverage

Intended branches

☑ Check burn executes successfully (full liquidity, partial liquidity multiple times, part of the minted liquidity).

Negative behavior

☑ Revert if `liq_nfts` is empty.
☑ Revert if emergency set.
☑ Revert if the token creator is wrong.
☑ Revert if the token collection name is wrong.

Function Call Analysis

- `burn -> assert_no_emergency()`
    - **External/Internal**: Internal.
    - **Argument control**: No.
    - **Impact**: Aborts if emergency set.
- `burn -> unwrap_liq_nft()`

- **External/Internal**: Internal.
  - **Argument control**: No.
  - **Impact**: Unwraps token ID, amount, and bin ID.
- `burn -> burn_bin_liquidity(pool, shares_amount, bin_id)`
  - **External/Internal**: Internal.
  - **Argument control**: Partial (amount controlled by input).
  - **Impact**: Withdraws X and Y reserves.
- `burn -> token::burn_by_creator(token_owner_account,@liquidswap_v1_resource pool.collection_name, token_name, 0,shares_amount)`
  - **External/Internal**: External.
  - **Argument control**: Partial (amount controlled by input).
  - **Impact**: Burns LB tokens.

Function: `flashloan<X, Y, BinStep>(amount_x: u64, amount_y: u64)`

This loans specified amounts of X and Y coins from the pool.

Inputs

- `amount_x`
  - **Validation**: Should be larger than zero.
  - **Impact**: Amount of X coins to loan.
- `amount_y`
  - **Validation**: Should be larger than zero.
  - **Impact**: Amount of Y coins to loan.

Branches and code coverage

Intended branches

- [x] `flashloan` executes successfully with no emergency set.

Negative behaviour

- [x] `flashloan` reverts if emergency set.
- [ ] `flashloan` reverts if `amount_x` or `amount_y` is zero.
- [ ] `flashloan` reverts if pool is locked.

Function Call Analysis

- `flashloan -> assert_no_emergency()`
  - **External/Internal**: Internal.

    - **Argument control**: No.
    - **Impact**: Reverts if emergency set.
- `flashloan -> coin::extract(coin, amount)`
    - **External/Internal**: External.
    - **Argument control**: Partial (controlled by `amount_x` and `amount_y`).
    - **Impact**: Amount of coins extracted from reserves.

Function: `mint<X, Y, BinStep>(coins_x: Coin<X>, coins_y: Coin<Y>, bin_ids: vector<u32>, liq_x_coins: vector<u64>, liq_y_coins: vector<u64>)`

This mints LB tokens by depositing X and Y coins into specified bins.

Inputs

- `coins_x`
    - **Validation**: None.
    - **Impact**: X coins to deposit.
- `coins_y`
    - **Validation**: None.
    - **Impact**: Y coins to deposit.
- `bin_ids`
    - **Validation**: Check that length matches other vectors.
    - **Impact**: Bins to deposit to.
- `liq_x_coins`
    - **Validation**: Check length matches other vectors.
    - **Impact**: Max X amounts for each bin.
- `liq_y_coins`
    - **Validation**: Check length matches other vectors.
    - **Impact**: Max Y amounts for each bin.

Branches and code coverage

Intended branches

- [x] Check mint executes successfully.

Negative behavior

- [x] Revert if emergency set.

Function Call Analysis

- `mint -> assert_no_emergency()`
  - **External/Internal**: Internal.
  - **Argument control**: No.
  - **Impact**: Aborts if emergency set.
- `mint -> coin::merge(coins, amount)`
  - **External/Internal**: External.
  - **Argument control**: Partial (controlled by input amounts).
  - **Impact**: Deposits X and Y coins into reserves.

Function: `mint_bins()`

This is an inner function that handles depositing into bins and minting LB tokens.

Inputs

- `pool`
  - **Validation**: None.
  - **Impact**: Provides access to pool state for deposits and minting.
- `received_x`
  - **Validation**: None.
  - **Impact**: Total X coins received.
- `received_y`
  - **Validation**: None.
  - **Impact**: Total Y coins received.
- `bin_ids`
  - **Validation**: None.
  - **Impact**: Bins to deposit to.
- `liq_x_coins`
  - **Validation**: None.
  - **Impact**: Max X amounts for each bin.
- `liq_y_coins`
  - **Validation**: None.
  - **Impact**: Max Y amounts for each bin.

Branches and code coverage

Intended branches

- ☑ Check mint executes successfully.

Negative behavior

- [x] Revert if emergency set.

Function Call Analysis

- `mint_bins -> update_bin()`
    - **External/Internal**: Internal.
    - **Argument control**:
        - `bin_id` — controlled by caller via `bin_ids`.
        - `total_supply` — not controlled.
        - `bin_step` — not controlled.
        - `active_bin_id` — not controlled.
        - `max_in_x` — controlled by caller via `liq_x_coins`.
        - `max_in_y` — controlled by caller via `liq_y_coins`.
    - **Impact**: Updates bin state and get deposits and shares for bin.
- `update_bin -> get_shares_and_amounts()`
    - **External/Internal**: Internal.
    - **Argument control**:
        - `reserves_x` — not controlled.
        - `reserves_y` — not controlled.
        - `in_x` — partially controlled via `max_in_x`.
        - `in_y` — partially controlled via `max_in_y`.
        - `price_fp64` — not controlled.
        - `total_supply` — not controlled.
    - **Impact**: Calculates deposits and shares for bin.
- `update_bin -> charge_composition_fees()`
    - **External/Internal**: Internal.
    - **Argument control**:
        - `reserves_x` — partially controlled via deposits.
        - `reserves_y` — partially controlled via deposits.
        - `fee_params` — not controlled.
        - `in_x` — partially controlled via `max_in_x`.
        - `in_y` — partially controlled via `max_in_y`.
        - `total_supply` — not controlled.
        - `shares` — partially controlled via deposits.
    - **Impact**: Charges composition fees for active bin.

Function: `pay_flashloan()`

This pays for the flash loan.

Inputs

- `coin_x_loan`
  - **Validation**: Check `coin_x_loan >= loan_x + lp_fee_x .`
  - **Impact**: X coins to pay loan — leftover used for fees.
- `coin_y_loan`
  - **Validation**: Check `coin_y_loan >= loan_y + lp_fee_y`.
  - **Impact**: Y coins to pay loan — leftover used for fees.
- `flashloan`
  - **Validation**: None.
  - **Impact**: Contains original loan amounts.

Branches and code coverage

Intended branches

- ☑ `pay_flashloan` executes successfully, and all coins are correctly paid in normal condition and the condition of overpayment.

Negative behavior

- ☑ `pay_flashloan` reverts if payment is insufficient.
- ☑ `pay_flashloan` reverts if pool is locked.

Function call analysis

- `pay_flashloan -> coin::extract(coin, amount)`
  - **External/Internal**: External.
  - **Argument control**: Partial (controlled by original loan amounts).
  - **Impact**: Extracts base loan payment into reserves.
- `pay_flashloan -> treasury::deposit(coin_x_loan, coin_y_loan)`
  - **External/Internal**: External.
  - **Argument control**: Controlled by the arguments of `pay_flashloan`.
  - **Impact**: Sends protocol fees to treasury.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed.

During our assessment on the scoped Liquidswap modules, we discovered two findings. No critical issues were found. One finding was of medium impact and the other finding was informational in nature. Pontem Network acknowledged all findings.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.