Pontem Network - Liquidswap DEX Flash Loans

Move Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **September 23rd, 2022 - September 27th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	09/23/2022	Lukasz Mikula
0.2	Draft Version	09/27/2022	Lukasz Mikula
0.3	Draft Review	09/29/2022	Gabi Urrutia
1.0	Remediation Plan	09/30/2022	Lukasz Mikula
1.1	Remediation Plan Review	09/30/2022	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Luis Quispe Gonzales	Halborn	Luis.QuispeGonzales@halborn.com
Lukasz Mikula	Halborn	Lukasz.Mikula@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Pontem Network engaged Halborn to conduct a security audit on their smart contracts beginning on September 23rd, 2022 and ending on September 27th, 2022 . The security assessment was scoped to the smart contracts provided in the GitHub repository Liquidswap, commit hashes and further details can be found in the Scope section of this report.

1.2 AUDIT SUMMARY

The team at Halborn was provided five days for the engagement and assigned one full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn found the contract to follow secure development best practices, resulting in only an informational finding with negligible security impact.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walk-through to identify any logic issue.
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Test coverage review (aptos move test).

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

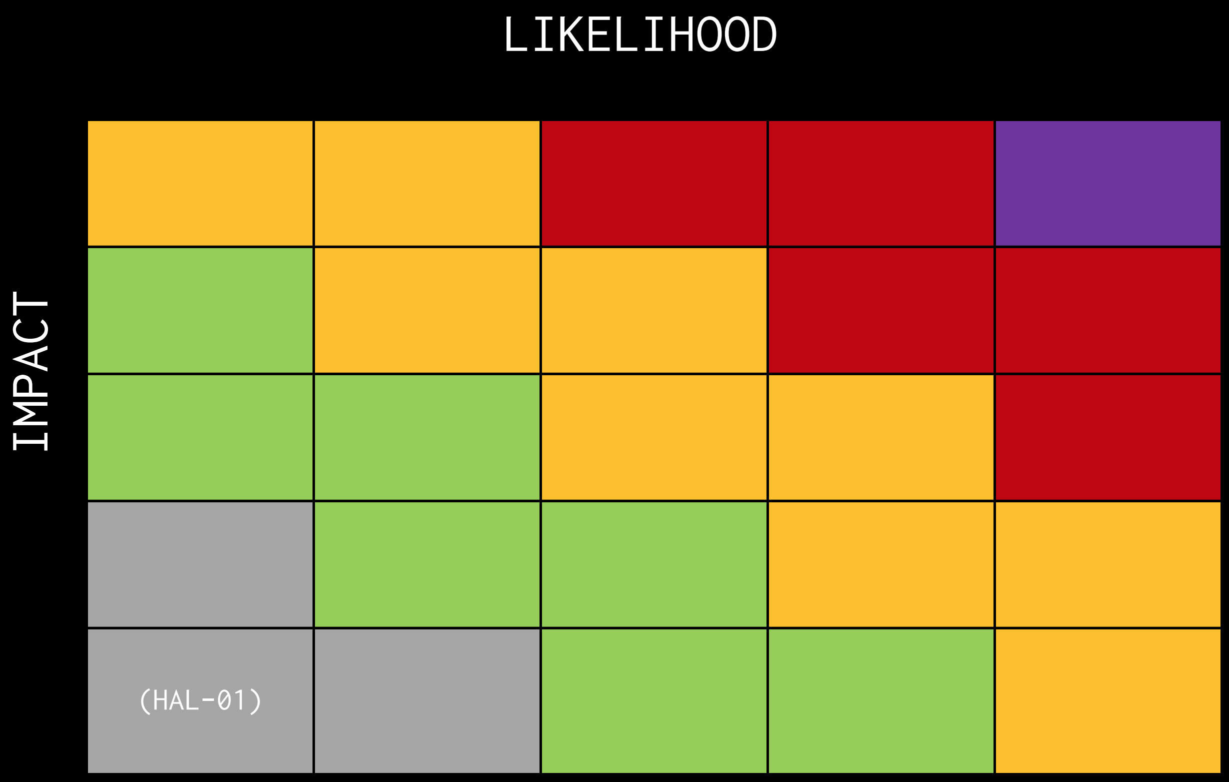
1.4 SCOPE

1. Move Smart Contract
   - (a) Repository: liquidswap
   - (b) Commit ID: 37705a42d8962a36472e4ae3ba93fb1bb38bdb49
   - (c) Contracts in scope:
     - liquidity_pool.move
   - (d) Functions in scope:
     - flashloan
     - pay_flashloan
     - all variables and subfunctions they utilize

Out-of-scope: External libraries and financial related attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	1



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) UNFINISHED CODE	Informational	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) UNFINISHED CODE - INFORMATIONAL

Description:

The code does not have available entry function. While the current implementation of flashloan and pay_flashloan was checked, it does not guarantee that a future entry point function will not contain potential issues.

Code Location:

Listing 1: liquidswap/sources/swap/liquidity_pool.move

```
    public fun flashloan<X, Y, Curve>(x_loan: u64, y_loan: u64): (Coin<X>, Coin<Y>, Flashloan<X, Y, Curve>)
    acquires LiquidityPool, EventsStore {
        assert_no_emergency();

        assert!(coin_helper::is_sorted<X, Y>(), ERR_WRONG_PAIR_ORDERING);
        assert!(exists<LiquidityPool<X, Y, Curve>>(@liquidswap_pool_account), ERR_POOL_DOES_NOT_EXIST);

        assert_pool_unlocked<X, Y, Curve>();

        assert!(x_loan > 0 || y_loan > 0, ERR_EMPTY_COIN_LOAN);

        let pool = borrow_global_mut<LiquidityPool<X, Y, Curve>>(@liquidswap_pool_account);

        let reserve_x = coin::value(&pool.coin_x_reserve);
        let reserve_y = coin::value(&pool.coin_y_reserve);

        // Withdraw expected amount from reserves.
        let x_loaned = coin::extract(&mut pool.coin_x_reserve, x_loan);
        let y_loaned = coin::extract(&mut pool.coin_y_reserve, y_loan);

        // The pool will be locked after the loan until payment.
        pool.locked = true;

        let events_store = borrow_global_mut<EventsStore<X, Y, Curve>>(@liquidswap_pool_account);
        event::emit_event(
            &mut events_store.loan_handle,
            FlashloanEvent<X, Y, Curve> {
                x_loan,
                y_loan,
            });

        update_oracle(pool, reserve_x, reserve_y);

        // Return loaned amount.
        (x_loaned, y_loaned, Flashloan<X, Y, Curve> {
            pool_addr: @liquidswap_pool_account,
            x_loan,
            y_loan,
        })
    }

    /// Pay flash loan coins.
    /// In the most of situation only X or Y coin argument has value.
    /// Because an user usually loans only one coin, yet function allow to loans both coin.
    /// * `x_in` - X coins to pay.
    /// * `y_in` - Y coins to pay.
    /// * `loan` - data about flashloan.
    /// Returns both loaned X and Y coins: `(Coin<X>, Coin<Y>, Flashloan<X, Y)`.
    public fun pay_flashloan<X, Y, Curve>(
        x_in: Coin<X>,
        y_in: Coin<Y>,
        loan: Flashloan<X, Y, Curve>
    ) acquires LiquidityPool {
        assert_no_emergency();

        assert!(coin_helper::is_sorted<X, Y>(), ERR_WRONG_PAIR_ORDERING);
        assert!(exists<LiquidityPool<X, Y, Curve>>(@liquidswap_pool_account), ERR_POOL_DOES_NOT_EXIST);

        let Flashloan { pool_addr, x_loan, y_loan } = loan;

        let x_in_val = coin::value(&x_in);
        let y_in_val = coin::value(&y_in);

        assert!(x_in_val > 0 || y_in_val > 0, ERR_EMPTY_COIN_IN);

        let pool = borrow_global_mut<LiquidityPool<X, Y, Curve>>(pool_addr);

        let x_reserve_size = coin::value(&pool.coin_x_reserve);
        let y_reserve_size = coin::value(&pool.coin_y_reserve);

        // Reserve sizes before loan out
        x_reserve_size = x_reserve_size + x_loan;
        y_reserve_size = y_reserve_size + y_loan;

        // Deposit new coins to liquidity pool.
        coin::merge(&mut pool.coin_x_reserve, x_in);
        coin::merge(&mut pool.coin_y_reserve, y_in);

        // Confirm that lp_value for the pool hasn't been reduced.
        // For that, we compute lp_value with old reserves and lp_value with reserves after swap is done,
        // and make sure lp_value doesn't decrease
        let (x_res_new_after_fee, y_res_new_after_fee) =
            new_reserves_after_fees_scaled<Curve>(
                coin::value(&pool.coin_x_reserve),
                coin::value(&pool.coin_y_reserve),
                x_in_val,
                y_in_val,
            );
        assert_lp_value_is_increased<Curve>(
            pool.x_scale,
            pool.y_scale,
            (x_reserve_size as u128),
            (y_reserve_size as u128),
            x_res_new_after_fee,
            y_res_new_after_fee,
        );
        // third of all fees goes into DAO
        split_third_of_fee_to_dao(pool, x_in_val, y_in_val);

        // As we are in same block, don't need to update oracle, it's already updated during flashloan initalization.

        // The pool will be unlocked after payment.
        pool.locked = false;
    }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to test the future implementation of entry function that will expose flashloans to users.

Remediation Plan:

ACKNOWLEDGED: The Pontem Network team acknowledged this finding.

THANK YOU FOR CHOOSING

HALBORN