Pulse X Stable Swap Security Audit

Report Version 1.0

July 29, 2024

Conducted by **Hunter Security**:

Stormy, Senior Security Auditor
Bounty Hunter, Senior Security Auditor
byter0x1, Senior Security Auditor
Zanderbyte, Junior Security Auditor
Amar Fares, Junior Security Auditor

Table of Contents

1 About Hunter Security

Hunter Security consists of multiple teams of leading smart contract security researchers. Having conducted over 100 security reviews and reported tens of live smart contract security vulnerabilities protecting over $1B of TVL, our team always strives to deliver top-quality security services to DeFi protocols. For security audit inquiries, you can reach out to him on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Hunter Security was engaged by PulseChain to review the Pulse X Stable Swap smart contract protocol during the period from July 1, 2024, to July 14, 2024.

Overview

Project Name	Pulse X Stable Swap
Repository	https://github.com/NinjaDev36/stable-swap
Commit hash	80e7b89e8754d5cac1c033e236d09c510abccbc7
Resolution	729de50d5919bc23906249b002aed416cb2c542a
Methods	Manual review & testing

Scope

contracts/PulseXStableSwapFactory.sol
contracts/PulseXStableSwapLP.sol
contracts/PulseXStableSwapLPFactory.sol
contracts/PulseXStableSwapThreePool.sol
contracts/PulseXStableSwapThreePoolDeployer.sol
contracts/PulseXStableSwapTwoPool.sol
contracts/PulseXStableSwapTwoPoolDeployer.sol
contracts/utils/*

Issues Found

High risk	0
Medium risk	5
Low risk	2
Informational	5

5 Findings

5.1 Medium

5.1.1 Adding partial liquidity may be bricked for certain pools

Severity: Medium

Context: PulseXStableSwapTwoPool.sol, PulseXStableSwapThreePool.sol

Description: If we look at the snippet below which is taken from the function *add_liquidity*, we can see that the system allows adding liquidity just for the tokens you want in the pool when the *token_supply* is above zero. Because of that the *amounts* array will have zero values for the tokens not provided.

```
for (uint256 i = 0; i < N_COINS; i++) {
    if (token_supply == 0) {
        require(amounts[i] > 0, "Initial deposit requires all coins");
    }
    // balances store amounts of c-tokens
    new_balances[i] = old_balances[i] + amounts[i];
}
```

This can be problematic as the code performs a transfer for every coin even if the amount is zero. The possibility of adding liquidity for just the tokens you want may be bricked if the particular token reverts on zero-value transfer. Currently, there is no restriction for the tokens used in the pools so the chance of this kind of token being used is decent.

```
for (uint256 i = 0; i < N_COINS; i++) {
    uint256 amount = amounts[i];
    address coin = coins[i];
    transfer_in(coin, amount);
}
```

Recommendation: Consider refactoring the function *transfer_in* and only doing transfers on values larger than zero:

```
function transfer_in(address coin_address, uint256 value) internal {
    if (coin_address == PLS_ADDRESS) {
        require(value == msg.value, "Inconsistent quantity");
    } else { // @audit add the following check:
        if (value != 0) IERC20(coin_address).safeTransferFrom(msg.sender, address(
            this), value);
    }
}
```

Resolution: Resolved.

5.1.2 Helper contracts won't work for a pool with both WPLS and PLS

Severity: Medium

Context: PulseXStableSwapTwoPoolWPLSHelper.sol, PulseXStableSwapThreePoolWPLSHelper.sol

Description: Take as an example, that a pool was created which uses both WPLS and PLS as tokens. *PulseXStableSwapThreePoolWPLSHelper* is a contract particularly made for supporting pools which have WPLS as tokens. However, the system won't fully work in this particular case.

The function *add_liquidity* is an external payable function which deposits the *msg.value* in order to gain WPLS tokens which are later transferred to the pool contract, as the *msg.value* is used only for gaining WPLS tokens as a result a user won't be able to add liquidity for PLS if they want to.

```
function add_liquidity(
    IPulseXStableSwap swap,
    uint256[N_COINS] memory amounts,
    uint256 min_mint_amount
) external payable {
    if (!isWhitelist[address(swap)]) revert NotWhitelist();
    if (swap.N_COINS() != N_COINS) revert InvalidNCOINS();
    if (!isApproved[address(swap)]) initSwapPool(swap);

    address token0 = swap.coins(0);
    address token1 = swap.coins(1);
    uint256 WPLSIndex;
    if (token0 == address(WPLS)) {
        WPLSIndex = 0;
    } else if (token1 == address(WPLS)) {
        WPLSIndex = 1;
    } else {
        revert NotWPLSPool();
    }
    require(msg.value == amounts[WPLSIndex], "Inconsistent quantity");
    WPLS.deposit{value: msg.value}();
```

Recommendation: Consider checking whether the pool is supporting the PLS address and reverting if that's the case. A separate function should be made if the system wants to handle a pool with both WPLS and PLS as tokens.

```
function add_liquidity(
    IPulseXStableSwap swap,
    uint256[N_COINS] memory amounts,
    uint256 min_mint_amount
) external payable {
    if (!isWhitelist[address(swap)]) revert NotWhitelist();
    if (swap.N_COINS() != N_COINS) revert InvalidNCOINS();
    if (!isApproved[address(swap)]) initSwapPool(swap);
    if (swap.support_PLS()) revert NotAllowed(); // @audit add this check
```

Resolution: Resolved.

5.1.3 Adding liquidity using the ThreePool helper contract will always revert

Severity: Medium

Context: PulseXStableSwapThreePoolWPLSHelper.sol

Description: A mistake was made which fetches the second token of the pool two times instead of the third one. In this case the system will provide wrong information to the swap contract and revert

duo to lack of funds for third token as only pool *token0* and *token1* were transferred from the user to the helper contract.

```
function add_liquidity(
    IPulseXStableSwap swap,
    uint256[N_COINS] memory amounts,
    uint256 min_mint_amount
) external payable {
    if (!isWhitelist[address(swap)]) revert NotWhitelist();
    if (swap.N_COINS() != N_COINS) revert InvalidNCOINS();
    if (!isApproved[address(swap)]) initSwapPool(swap);

    address token0 = swap.coins(0);
    address token1 = swap.coins(1);
    address token2 = swap.coins(1); // @audit should be 2
```

Recommendation: Consider implementing the following change:

```
function add_liquidity(
    IPulseXStableSwap swap,
    uint256[N_COINS] memory amounts,
    uint256 min_mint_amount
) external payable {
    if (!isWhitelist[address(swap)]) revert NotWhitelist();
    if (swap.N_COINS() != N_COINS) revert InvalidNCOINS();
    if (!isApproved[address(swap)]) initSwapPool(swap);

    address token0 = swap.coins(0);
    address token1 = swap.coins(1);
    // address token2 = swap.coins(1); // @audit remove
    address token2 = swap.coins(2); // @audit add
```

Resolution: Resolved.

5.1.4 No functionality to withdraw excess fees after the pool is killed

Severity: Medium

Context: PulseXStableSwapTwoPool.sol, PulseXStableSwapThreePool.sol

Description: The internal function *_withdraw_admin_fees* is used when adding or removing liquidity from the pool contract. The purpose of this function is to withdraw any excess funds which are not part of pool balances. The problem here is that after a pool is killed most of its core functions are not functioning anymore and only removing liquidity from the pool is allowed. Take as an example that there is no more liquidity in the pool, but there are excess funds in the contracts which needs to be retrieved. In this case there is no option to withdraw them as *_withdraw_admin_fees* is internal.

```
function _withdraw_admin_fees() internal {
    address fee_to = IPulseXStableSwapFactory(STABLESWAP_FACTORY).fee_to();
    if (fee_to == address(0)) return;

    for (uint256 i = 0; i < N_COINS; i++) {
        uint256 value;
        // check for fee amounts, underflow safe
        if (coins[i] == PLS_ADDRESS) {
```

```
            uint256 balance = address(this).balance;
            value = balance > balances[i] ? balance - balances[i] : 0;
            if (value > 0) {
                IWPLS(WPLS).deposit{ value: value }();
                IWPLS(WPLS).transfer(fee_to, value);
            }
        } else {
            uint256 balance = IERC20(coins[i]).balanceOf(address(this));
            value = balance > balances[i] ? balance - balances[i] : 0;
            if (value > 0) {
                IERC20(coins[i]).safeTransfer(fee_to, value);
            }
        }
    }
}
```

Recommendation: Apart from the internal function *_withdraw_admin_fees*, it would be beneficial to make identical *onlyOwner* function which can be called only when the pool is killed. This way if the pool is killed and there is no one to remove liquidity, the admin fees can still be withdrawn.

Resolution: Resolved.

5.1.5 donate_admin_fees can break the invariant check for initial deposit

Severity: Medium

Context: PulseXStableSwapTwoPool.sol, PulseXStableSwapThreePool.sol

Description: When adding liquidity an important invariant is created which stands that the first initial deposit should be for all the tokens in the pool otherwise further issues can occur when removing or exchanging when the balance for a particular pool token is zero.

```
for (uint256 i = 0; i < N_COINS; i++) {
    if (token_supply == 0) {
        require(amounts[i] > 0, "Initial deposit requires all coins");
    }
    // balances store amounts of c-tokens
    new_balances[i] = old_balances[i] + amounts[i];
}
```

The function *donate_admin_fees* is used by the owner in order to donate the excess funds in the contract which were previously taken as an admin fee. The problem here is that the system doesn't check if the overriden balance of a token is zero after donation, in this case a donation on empty balances may bypass the initial deposit variant which is enforced when adding liquidity to the pool.

```
function donate_admin_fees() external onlyOwner {
    for (uint256 i = 0; i < N_COINS; i++) {
        if (coins[i] == PLS_ADDRESS) {
            balances[i] = address(this).balance;
        } else {
            balances[i] = IERC20(coins[i]).balanceOf(address(this));
        }
    }
    emit DonateAdminFees();
}
```

Recommendation: Consider making sure that all the token balances are positive after donation otherwise the initial deposit variant could be broken:

```
function donate_admin_fees() external onlyOwner {
    for (uint256 i = 0; i < N_COINS; i++) {
        if (coins[i] == PLS_ADDRESS) { // @audit add below check
            if (address(this).balance == 0) revert NotAllowed();
            balances[i] = address(this).balance;
        } else { // @audit add below check
            if (IERC20(coins[i]).balanceOf(address(this)) == 0) revert NotAllowed();
            balances[i] = IERC20(coins[i]).balanceOf(address(this));
        }
    }
    emit DonateAdminFees();
}
```

Resolution: Resolved.

5.2 Low

5.2.1 Protocol does not support fees on transfer tokens

Severity: Low

Context: PulseXStableSwapTwoPool.sol, PulseXStableSwapThreePool.sol

Description: The current implementation of the *TwoPool* and *ThreePool* does not support tokens that charge fees on transfer. In case these tokens are used, the amount transferred in and out of the protocol will be inaccurately calculated.

Recommendation: Only allow the use of tokens that does not charge fees, if this is intended. Otherwise, consider modifying the code to accommodate the fee on transfer tokens.

Resolution: Acknowledged.

5.2.2 More than one pool can be created for a token pair

Severity: Low

Context: PulseXStableSwapFactory.sol, PulseXStableSwapTwoPoolDeployer.sol, PulseXStableSwapThreePoolDeployer.sol

Description: The *createTwoPool()* and *createThreePool()* functions call the *createPool()* function on the deployer contracts to create pools for pairs of two and three tokens. The latter function creates a salt that is used in the *create2()* operation and one of the values used in generating the salt hash is the *block.timestamp*.

The use of the block timestamp in the salt allows for the creation of different salts for the same token pair, allowing the admin to create more than one pool for the same set of tokens in a token pair.

Recommendation: This could be intended behavior, so the logic should be left as is if that is the case. Otherwise, remove *block.timestamp* from the hash generation logic, so that an address conflict occurs if the same pool is being created again, causing a revert.

Resolution: Acknowledged.

5.3 Informational

5.3.1 Unnecessary use of sorting logic

Severity: Informational

Context: PulseXStableSwapTwoPoolDeployer.sol, PulseXStableSwapThreePoolDeployer.sol

Description: The *createPool()* function in the two contracts calls *sortTokens()*. However, the call to *createPool()* is always initiated by the *PulseXStableSwapFactory*, which does the sorting before the function call to create the pool with the values already sorted as *tokenA*, *tokenB* and *tokenC* (in the case of a three pair pool).

Recommendation: Remove the *sortTokens()* function from the deployer contracts, along with its calls in the *createPool()* logic.

Resolution: Resolved.

5.3.2 Lack of zero address checks

Severity: Informational

Context: PulseXStableSwapFactory.sol

Description: The code contains multiple instances where a user supplied address is assigned as the value of a state variable without validating that the value is not the zero address.

Recommendation: Check against the zero address before assigning the values.

Resolution: Resolved.

5.3.3 Owner can add info of pools not created by factory

Severity: Informational

Context: PulseXStableSwapFactory.sol

Description: The *addPoolInfo()* function allows the contract owner to add pool information for any pool. However, the current implementation allows the owner to add information for any contract that matches the *IPulseXStableSwap* interface, even if it was not created by the factory contract.

Recommendation: Consider removing the function since pools will only be created via the factory and their information will be recorded at deployment time and will never need to be updated.

Resolution: Acknowledged.

5.3.4 Contract can have no owner

Severity: Informational

Context: PulseXStableSwapFactory.sol, PulseXStableSwapTwoPoolWPLSHelper.sol, PulseXStableSwapThreePoolWPLSHelper.sol

Description: The *renounceOwnership()* function in the contracts listed below is not overridden with reverting logic. This means the contract owners can call the function and leave the contracts with no owners, rendering all *onlyOwner* protected functions non-callable.

Recommendation: Override the *renounceOwnership()* function and add logic to revert.

Resolution: Resolved.