TSwap Audit Report

Version 1.0

Blurdragon101

April 21, 2024

TSwap Audit Report

Bluedragon101

April 21, 2024

Prepared by: Bluedragon101 Lead Auditors:

- [Shibi_Kishore]

Table of Contents

- Medium
  * [M-1] `TSwapPool::deposit` is missing deadline check causing transcations to complete even after the deadline
  * [M-2] Rebase, fee-on transfer and ERC 777 tokens, causes the protocol invariant to break
- Low
  * [L-1] `TSwapPool::LiquidityAdded` event is out of order
  * [L-2] Default value returned in `TSwapPool::swapExactInput` results in incorrect return value given
- Informationals
  * [I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` error is not used and should be removed
  * [I-2] Lack of zero address check in `PoolFactory::contstructor`
  * [I-3] `PoolFactory::liquidityTokenSymbol` should use `.symbol()` instead of `.name()`
  * [I-4] Event is missing `indexed` fields

Disclaimer

The Bluedragon101 team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Protocol Summary

TSwap

This project is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX). T-Swap is known as an Automated Market Maker (AMM) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset.

Core Invariant

Our system works because the ratio of Token A & WETH will always stay the same. Well, for the most part. Since we add fees, our invariant technially increases.

`x * y = k`

- x = Token Balance X
- y = Token Balance Y
- k = The constant ratio between X & Y

Our protocol should always follow this invariant in order to keep swapping correctly!

Audit Details

The findings described in this document correspond the following commit hash:

```
e643a8d4c2c802490976b538dd009b351b1c8dda
```

Scope

```
./src/
#-- PoolFactory.sol
#-- TSwapPool.sol
```

Roles

- Liquidity Providers: Users who have liquidity deposited into the pools. Their shares are represented by the LP ERC20 tokens. They gain a 0.3% fee every time a swap is made.
- Users: Users who want to swap tokens.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	4
Total	12

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost of fees

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user should deposit given an amount of tokens of output tokens. However, the function currently miscalculates the amount of tokens to take from the user. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users

Proof of Concept:

1. First liquidity provider deposits 100 WETH and 100 pool token to the pool.
2. User has 0 WETH and 11 pool token.
3. User wants to swap 1 WETH to pool token.
4. User calls `swapExactOutput` function.
5. As there is wrong fee calculation, user will get less pool token than expected.

Proof Of Code:

```
function test_FlawedSwapExactOutput() public {
        // Lets first deposit some liquidity
        uint256 initialLiquidty = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(initialLiquidty, 0, initialLiquidty, uint64(block.
            timestamp));
        vm.stopPrank();

        console.log("Pool WETH balance: ", weth.balanceOf(address(pool)
            ));
        console.log("Pool Token balance: ", poolToken.balanceOf(address
            (pool)));
        console.log("The balance of pool token of liquidity provider: "
            , poolToken.balanceOf(liquidityProvider));

        // Now lets say User has 11 pool tokens
        uint256 userInitialPoolTokens = 11e18;
        poolToken.mint(user, userInitialPoolTokens);
        console.log("User's pool token balance: ", poolToken.balanceOf(
            user));
        vm.startPrank(user);

        // User wants to buy 1 WETH from the pool, paying with pool
            tokens
        // As the initial is 1:1, so user should have paid ~ 1 pool
            token
        poolToken.approve(address(pool), type(uint256).max);
        uint256 actualValue = pool.swapExactOutput(poolToken, weth, 1
            ether, uint64(block.timestamp));
        console.log("Actual value: ", actualValue);
        console.log("User's pool token balance after swap: ", poolToken
            .balanceOf(user));
        vm.stopPrank();

        console.log("Pool WETH balance after swap: ", weth.balanceOf(
            address(pool)));
        console.log("Pool Token balance after swap: ", poolToken.
            balanceOf(address(pool)));
        console.log("The total supply of pool tokens: ", pool.
            totalSupply());

        // Now lets calculate the expected value
        // The formula is: x * y = k
        // ((100e18 * 1e18) * 10_000) / ((100e18 - 1e18) * 997)) =~
            1.003e19

        // lets say we are swaping USDC and WETH
        // Means for 1 Weth we need to pay 10.003 USDC intead of 1 USDC

        // Recommeded to use this formuala
        // ((100e18 * 1e18) * 1_000) / ((100e18 - 1e18) * 997)) =~
            1.003e18

        // So a liquidity provider can rug pull all the liquidity from
```

```
            the pool
        vm.startPrank(liquidityProvider);
        pool.withdraw(pool.balanceOf(liquidityProvider), 1, 1, uint64(
            block.timestamp));
        vm.stopPrank();

        console.log("Pool WETH balance after rug pull: ", weth.
            balanceOf(address(pool)));
        console.log("Pool Token balance after rug pull: ", poolToken.
            balanceOf(address(pool)));
    }
```

Foundry's output:

```
Logs:
  Pool WETH balance:  100000000000000000000
  Pool Token balance:  100000000000000000000
  The balance of pool token of liquidity provider:
      100000000000000000000
  Users pool token balance:  11000000000000000000
  Actual value:  10131404313951956880
  Users pool token balance after swap:  868595686048043120
  Pool WETH balance after swap:  99000000000000000000
  Pool Token balance after swap:  110131404313951956880
  The total supply of pool tokens:  100000000000000000000
  Pool WETH balance after rug pull:  0
  Pool Token balance after rug pull:  0
```

Recommended Mitigation:

```
function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-        return ((inputReserves * outputAmount) * 10_000) / ((
    outputReserves - outputAmount) * 997);
+        return ((inputReserves * outputAmount) * 1_000) / ((
    outputReserves - outputAmount) * 997);
    }
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` results in users receiving less tokens than expected

Description: The `swapExactOutput` function does not include any sort of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput` where the user specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount` to protect the user from receiving less tokens than expected.

Impact: If market conditions change during the transaction processess, the user could get a much worse swap.

Proof of Concept:

1. The price of WETH right now is 1,000 USDC
2. Users inputs a `swapExactOutput` transaction to swap 1 WETH
    1. input token = USDC
    2. output token = WETH
    3. output Amount = 1 WETH
    4. deadline = whatever
3. The function does not offer a maxInput amount
4. As the transcation is pending in mempool, the market changes and the price moves huge -> 1 WETH is now 10,000 USDC, 10 times more expensive
5. The transaction is executed, but the user receives 0.1 WETH instead of 1 WETH and spends 10,000 USDC instead of 1,000 USDC

Recommended Mitigation: We should include a `maxInputAmount` parameter in the `swapExactOutput` function, so the user only has to spend up to specific amount, and predict how much they will spend on the protocol.

```
function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
+       uint256 maxInputAmount,
        uint256 outputAmount,
        uint64 deadline
    )
        public
        revertIfZero(outputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 inputAmount)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        inputAmount = getInputAmountBasedOnOutput(outputAmount,
            inputReserves, outputReserves);
+       if(inputAmount > maxInputAmount) {
+           revert();
+       }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
    }
```

[H-3] `TSwapPool::sellPoolTokens` mismatches input and output token causing users to receive the incorrect amount of tokens

Description: The `sellPoolTokens` function is intended to allow user to sell pool tokens for WETH. Users indicate how many pool token they're willing to sell in the `poolTokenAmount` parameter. However, the fuction is currently miscalculates the swapped amount.

This is due to `swapExactOutput` function is called, whereas the `swapExactInput` function is the one should be called.Because users specify the exact amount of input tokens, not output tokens.

Impact: Users will swap the wrong amount of tokens, resulting in a loss of funds.

Proof of Concept:

Recommended Mitigation: Consider changing the implementation to `swapExactInput` instead of `swaoExactOutput`. Note that this would also require changin the `sellPoolTokens` function to accept a new parameter (ie `minWethToReceive` passed to `swapExactInput`).

```
function sellPoolTokens(
    uint256 poolTokenAmount
+   uint256 minWethToReceive
    ) external returns (uint256 wethAmount) {
-       return swapExactOutput(i_poolToken, i_wethToken,
    poolTokenAmount, uint64(block.timestamp));
+       return swapExactInput(i_poolToken, i_wethToken, poolTokenAmount
    , minWethToReceive, uint64(block.timestamp));
}
```

Additionally, it might be wise to a deadline to the function, as there is currently no deadline check.

[H-4] In `TSwapPool::_swap` the extra tokens given to users after every swapCount breaks the protocol invariant of `x * y = k`

Description: The protocol follows a strict invariant of `x * y = k` where `x` is the amount of input tokens, `y` is the amount of output tokens, and `k` is a constant. However, the `_swap` function gives extra tokens to users after every `swapCount` transactions. This breaks the invariant of protocol for every 10 `swapCount`.

The following block of code is responsible for the issue:

```
swap_count++;
if (swap_count >= SWAP_COUNT_MAX) {
  swap_count = 0;
  outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
}
```

Impact: A user could maliciously drain the protocol of funds by doing a lot of swaps and collecting the extra incentive given out of by the protocol.

Most simply put, the protocol's core invariant is broken.

Proof of Concept:

1. A user swaps 10 times, and collects the extra incentive of `1_000_000_000_000_000_000` tokens.
2. That user continues to swap untill the protocol is drained of funds.

Proof Of Code

```
function testInvariantBroken() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 outputWeth = 1e17;
        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
        for (uint256 i = 0; i < 10; i++) {
            pool.swapExactOutput(
                poolToken,
                weth,
                outputWeth,
                uint64(block.timestamp)
            );
        }
        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);
        assertEq(actualDeltaY, expectedDeltaY);
    }
```

Recommended Mitigation: Remove the extra incentive. If you want to keep this in, we should account for the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

```
-   swap_count++;
-            if (swap_count >= SWAP_COUNT_MAX) {
-                swap_count = 0;
-                outputToken.safeTransfer(msg.sender, 1
   _000_000_000_000_000_000);
-          }
```

Medium

[M-1] `TSwapPool::deposit` is missing deadline check causing transcations to complete even after the deadline

Description: The `deposit` function accepts a deadline parameter according to the documentation. However, the function does not check the deadline parameter. As a consequence, operations that add liquidity to the pool can be executed at unexpected times, which can lead to unexpected results.

Impact: Transactions could be sent when market condition is unfavorable to deposit, even when adding a deadline paramter.

Proof of Concept: The `deadline` parameter is not used in the `deposit` function.

Recommended Mitigation: Add a check for the deadline parameter in the `deposit` function.

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
+       revertIfDeadlinePassed(deadline)
        returns (uint256 liquidityTokensToMint)
    {
    .
    .
    .
    }
```

[M-2] Rebase, fee-on transfer and ERC 777 tokens, causes the protocol invariant to break

Description: The protocol follows a strict invariant of `x * y = k` where `x` is the amount of input tokens, `y` is the amount of output tokens, and `k` is a constant. However, the protocol does not account for rebasing tokens, fee-on transfer tokens, and ERC 777 tokens. This can cause the protocol to break the invariant.

Impact: The protocol invariant is broken, causing the protocol to lose funds.

Recommended Mitigation: The protocol should not allow rebasing tokens, fee-on transfer tokens, and ERC 777 tokens. The protocol should only allow ERC 20 tokens.

Low

[L-1] `TSwapPool::LiquidityAdded` event is out of order

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTrans`, the order of the parameters is not consistent with the order of the parameters in the function signature. This can make it difficult for off-chain tools to parse the event.

Impact: Event emission is incorrect, leading to offchain function parsing issues.

Proof of Concept: The `poolTokensDeposited` value should go in third parameter position and `wethDeposited` parameters value should go in second in the `LiquidityAdded` event.

Recommended Mitigation: Add the following

```diff
-    emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit)
    ;
+    emit LiquidityAdded(msg.sender, wethToDeposit, poolTokensToDeposit)
    ;
```

[L-2] Default value returned in `TSwapPool::swapExactInput` results in incorrect return value given

Description: The `swapExactInput` function is expected to return the actual amount of tokens received by the user. However, while it declares the named return value `output` it is never assigned value, nor uses explicit return statement. As a result, the function returns the default value of `output`, which is 0.

Impact: The return value of the `swapExactInput` function is always be 0, giving incorrect return value.

Proof of Concept:

Recommended Mitigation:

```
{
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

-        uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
    inputReserves, outputReserves);
+        output = getOutputAmountBasedOnInput(inputAmount, inputReserves
    , outputReserves);

-         if (outputAmount < minOutputAmount) {
-             revert TSwapPool__OutputTooLow(outputAmount,
    minOutputAmount);
-         }
+         if (output < minOutputAmount) {
+             revert TSwapPool__OutputTooLow(output, minOutputAmount);
+         }

-         _swap(inputToken, inputAmount, outputToken, outputAmount);
+         _swap(inputToken, inputAmount, outputToken, output);
    }
```

Informationals

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` error is not used and should be removed

Recommended Mitigation:

```
-    error PoolFactory__PoolDoesNotExist(address tokenAddress);
```

[I-2] Lack of zero address check in `PoolFactory::contstructor`

Recommended Mitigation:

```
constructor(address wethToken) {
+        if(wethToken == address(0)) {
+            revert("PoolFactory: wethToken is zero address");
        }
        i_wethToken = wethToken;
    }
```

[I-3] `PoolFactory::liquidityTokenSymbol` should use `.symbol()` instead of `.name()`

Recommended Mitigation:

```
-    string memory liquidityTokenSymbol = string.concat("ts", IERC20(
    tokenAddress).name());
+    string memory liquidityTokenSymbol = string.concat("ts", IERC20(
    tokenAddress).symbol());
```

[I-4] Event is missing `indexed` fields

Description: Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Recommended Mitigation:

Found in src/PoolFactory.sol Line: 35

```
-    event PoolCreated(address tokenAddress, address poolAddress);
+    event PoolCreated(address indexed tokenAddress, address indexed
    poolAddress);
```

Found in src/TSwapPool.sol Line: 43

```
-    event LiquidityAdded(address indexed liquidityProvider, uint256
    wethDeposited, uint256 poolTokensDeposited);
+    event LiquidityAdded(address indexed liquidityProvider, uint256
    indexed wethDeposited, uint256 indexed poolTokensDeposited);
```

Found in src/TSwapPool.sol Line: 44

```
-        event LiquidityRemoved(address indexed liquidityProvider,
    uint256 wethWithdrawn, uint256 poolTokensWithdrawn);
+        event LiquidityRemoved(address indexed liquidityProvider,
    uint256 indexed wethWithdrawn, uint256 indexed poolTokensWithdrawn);
```

Found in src/TSwapPool.sol Line: 45

```
-        event Swap(address indexed swapper, IERC20 tokenIn, uint256
    amountTokenIn, IERC20 tokenOut, uint256 amountTokenOut);
+        event Swap(address indexed swapper, IERC20 indexed tokenIn,
    uint256 amountTokenIn, IERC20 indexed tokenOut, uint256
    amountTokenOut);
```