TSwap Protocol Audit Report

Version 1.0

Peter Berekvolgyi

March 31, 2025

TSwap Protocol Audit Report

Peter Berekvolgyi

March 31, 2025

Prepared by: Peter Berekvolgyi

Table of Contents

- Medium
  - [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline
  - [M-2] Rebase and fee-on-transfer tokens break protocol invariant
- Low
  - [L-1] `TSwapPool::LiquidityAdded` event has parameters out of order
  - [L-2] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given
- Informational
  - [I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed
  - [I-2] Lacking zero address checks
  - [I-3] `PoolFacotry::createPool` should use `.symbol()` instead of `.name()`
  - [I-4] Poor test coverage

Protocol Summary

The protocol implements a permissionless decentralized exchange using the constant product formula.

Disclaimer

Peter Berekvolgyi 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 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

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

The CodeHawks severity matrix is used to determine severity. See the documentation for more details.

Audit Details

Scope

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

Roles

- Liquidity provider: Deposits pool tokens and WETH to the pools in exchange of shares from the fees. Can withdraw deposited liquidity.
- Swapper: Swaps tokens for WETH and vice-versa.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	4
Gas Optimizations	0

Severity	Number of issues found
Total	12

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost 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 resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

As a result, users swapping tokens via the `swapExactOutput` function will pay far more tokens than expected for their trades. This becomes particularly risky for users that provide infinite allowance to the `TSwapPool` contract. Moreover, note that the issue is worsened by the fact that the `swapExactOutput` function does not allow users to specify a maximum of input tokens, as is described in another issue in this report.

It's worth noting that the tokens paid by users are not lost, but rather can be swiftly taken by liquidity providers. Therefore, this contract could be used to trick users, have them swap their funds at unfavorable rates and finally rug pull all liquidity from the pool.

Impact: Protocol takes more fees than expected from users.

Proof of Concept:

```
function testFlawedSwapExactOutput() public {
    uint256 initialLiquidity = 100e18;
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), initialLiquidity);
    poolToken.approve(address(pool), initialLiquidity);

    pool.deposit({
        wethToDeposit: initialLiquidity,
        minimumLiquidityTokensToMint: 0,
        maximumPoolTokensToDeposit: initialLiquidity,
        deadline: uint64(block.timestamp)
    });
    vm.stopPrank();
```

```
14
15      // User has 11 pool tokens
16      address someUser = makeAddr("someUser");
17      uint256 userInitialPoolTokenBalance = 11e18;
18      poolToken.mint(someUser, userInitialPoolTokenBalance);
19      vm.startPrank(someUser);
20
21      // Users buys 1 WETH from the pool, paying with pool tokens
22      poolToken.approve(address(pool), type(uint256).max);
23      pool.swapExactOutput(
24          poolToken,
25          weth,
26          1 ether,
27          uint64(block.timestamp)
28      );
29
30      // Initial liquidity was 1:1, so user should have paid ~1 pool
            token
31      // However, it spent much more than that. The user started with 11
            tokens, and now only has less than 1.
32      assertLt(poolToken.balanceOf(someUser), 1 ether);
33      vm.stopPrank();
34
35      // The liquidity provider can rug all funds from the pool now,
36      // including those deposited by user.
37      vm.startPrank(liquidityProvider);
38      pool.withdraw(
39          pool.balanceOf(liquidityProvider),
40          1, // minWethToWithdraw
41          1, // minPoolTokensToWithdraw
42          uint64(block.timestamp)
43      );
44
45      assertEq(weth.balanceOf(address(pool)), 0);
46      assertEq(poolToken.balanceOf(address(pool)), 0);
47  }
```

Recommended Mitigation:

```
1       function getInputAmountBasedOnOutput(
2           uint256 outputAmount,
3           uint256 inputReserves,
4           uint256 outputReserves
5       )
6           public
7           pure
8           revertIfZero(outputAmount)
9           revertIfZero(outputReserves)
10          returns (uint256 inputAmount)
11      {
12  -           return ((inputReserves * outputAmount) * 10_000) / ((
```

```
        outputReserves - outputAmount) * 997);
13 +        return ((inputReserves * outputAmount) * 1_000) / ((
        outputReserves - outputAmount) * 997);
14      }
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

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 function specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact: If market conditions change before the transaciton processes, the user could get a much worse swap.

Proof of Concept: 1. The price of 1 WETH right now is 1,000 USDC 2. User inputs a `swapExactOutput` looking for 1 WETH 1. inputToken = USDC 2. outputToken = WETH 3. outputAmount = 1 4. deadline = whatever 3. The function does not offer a maxInput amount 4. As the transaction is pending in the mempool, the market changes! And the price moves HUGE -> 1 WETH is now 10,000 USDC. 10x more than the user expected 5. The transaction completes, but the user sent the protocol 10,000 USDC instead of the expected 1,000 USDC

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

```
1       function swapExactOutput(
2           IERC20 inputToken,
3  +        uint256 maxInputAmount,
4  .
5  .
6  .
7           inputAmount = getInputAmountBasedOnOutput(outputAmount,
               inputReserves, outputReserves);
8  +        if(inputAmount > maxInputAmount){
9  +            revert TSwapPool__OutputTooHigh(inputAmount, maxInputAmount
       );
10 +        }
11          _swap(inputToken, inputAmount, outputToken, outputAmount);
```

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

Description: The `sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. Users indicate how many pool tokens they're willing to sell in

the `poolTokenAmount` parameter. However, the function currently miscalculaes the swapped amount.

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

Impact: Users will swap the wrong amount of tokens, which is a severe disruption of protcol functionality.

Recommended Mitigation:

Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (ie `minWethToReceive` to be 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, poolTokenAmount, i_wethToken, minWethToReceive, uint64(block.timestamp));
    }
```

Additionally, it might be wise to add a deadline to the function, as there is currently no deadline. (MEV later)

[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`: The balance of WETH - `y`: The balance of the pool token - `k`: The constant product of the two balances

This means, that whenever the balances change in the protocol, the ratio between the two amounts should remain constant, hence the `k`. However, this is broken due to the extra incentive in the `_swap` function. Meaning that over time the protocol funds will be drained.

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);
```

```
5           }
```

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

Proof Of Code

Place the following into `TSwapPool.t.sol`.

```
1     function testInvariantBroken() public {
2         vm.startPrank(liquidityProvider);
3         weth.approve(address(pool), 100e18);
4         poolToken.approve(address(pool), 100e18);
5         pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
6         vm.stopPrank();
7
8         uint256 outputWeth = 1e17;
9
10        vm.startPrank(user);
11        poolToken.approve(address(pool), type(uint256).max);
12        poolToken.mint(user, 100e18);
13        for (uint256 i = 0; i < 9; i++) {
14            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                  block.timestamp));
15        }
16
17        int256 startingY = int256(weth.balanceOf(address(pool)));
18        int256 expectedDeltaY = int256(-1) * int256(outputWeth);
19
20        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
              timestamp));
21        vm.stopPrank();
22
23        uint256 endingY = weth.balanceOf(address(pool));
24        int256 actualDeltaY = int256(endingY) - int256(startingY);
25        assertEq(actualDeltaY, expectedDeltaY);
26    }
```

Recommended Mitigation: Remove the extra incentive mechanism. 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.

```
1 -         swap_count++;
2 -         // Fee-on-transfer
3 -         if (swap_count >= SWAP_COUNT_MAX) {
```

```
-            swap_count = 0;
-            outputToken.safeTransfer(msg.sender, 1
    _000_000_000_000_000_000);
-        }
```

By removing the above machanism, SWAP_COUNT_MAX constant also becomes unused, so remove that as well.

```
    IERC20 private immutable i_wethToken;
    IERC20 private immutable i_poolToken;
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
    uint256 private swap_count = 0;
-    uint256 private constant SWAP_COUNT_MAX = 10;
```

Medium

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

Description: The `deposit` function accepts a deadline parameter, which according to the documentation is "The deadline for the transaction to be completed by". However, this parameter is never used. As a consequence, operationrs that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavorable.

Impact: Transactions could be sent when market conditions are unfavorable to deposit, even when adding a deadline parameter.

Proof of Concept: The `deadline` parameter is unused.

Recommended Mitigation: Consider making the following change to the function.

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

[M-2] Rebase and fee-on-transfer tokens break protocol invariant

Description Rebase tokens break the assumption that balances remain constant. Fee-on-transfer tokens break the assumption of the other party receis the same amount of tokens that are being swapped.

Impact

Rebase tokens: any time a rebase happens the formula breaks, thus the pool's price becomes incorrect.

Fee-on-transfer tokens: the pool over- or underestimates received tokens and thus experiences high slippage. Also since the liquidity is decreasing over time, it causes impermanent loss.

Proof of Concept

Rebase tokens: at each rebase the total supply of tokens change, meaning that the balance of tokens also change in the pool without doing a swap. Mathematically `x * y = k` would become `x * (y + delta y)= k` which is not true any more because `x` has not changed.

Fee-on-transfer tokens: at each swap, liquidity provision and withdrawal the pool or user receives less tokens than expected because of the token fee. At swapping this invariant should stand: `x * y = (x + delta x)* (y * delta y)`, however, `delta y` would always be less because token fees are either burned or transferred and thus this invariant doesn't stand.

Recommended Mitigation

Either whitelist tokens (this could add centralization to the protocol or complex voting mechanisms) or keep an updated list of weird tokens and warn users about them on the web application.

Low

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

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran` function, it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the third parameter position, whereas the `wethToDeposit` value should go second.

Impact: Event emission is incorrect, leading to off-chain functions potentially malfunctioning.

Recommended Mitigation:

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

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

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return value `ouput` it is never assigned a value, nor uses an explicit return statement.

Impact: The return value will always be 0, giving incorrect information to the caller.

Recommended Mitigation:

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

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

Informational

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

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

[I-2] Lacking zero address checks

```
    constructor(address wethToken) {
+       if(wethToken == address(0)) {
```

```
+             revert();
+         }
          i_wethToken = wethToken;
      }
```

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

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

[I-4] Poor test coverage

		Coverage report		
File	% Lines	% Statements	% Branches	% Funcs
src/PoolFactory.sol	88.89% (16/18)	93.75% (15/16)	100.00% (1/1)	80.00% (4/5)
src/TSwapPool.sol	51.00% (51/100)	55.88% (57/102)	0.00% (0/13)	45.00% (9/20)