TSwap Protocol Audit Report

Version 1.0

Daniel Stamler

April 23, 2024

TSwap Protocol Audit Report

Daniel Stamler

April 23, 2024

Prepared by: Daniel Stamler - Independent Security Researcher

Table of Contents

Protocol Summary

The TSwap protocol consists of two main Solidity contracts: `PoolFactory` and `TSwapPool`. The `PoolFactory` acts as a central hub for creating decentralized liquidity pools, with each pool being instantiated through a distinct `TSwapPool` contract. These pools facilitate token exchanges between a pool's native token and WETH, while also allowing users to add or remove liquidity as needed. This system provides a robust platform for decentralized trading, accommodating various user interactions to manage liquidity efficiently

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

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

Audit Details

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

```
55d1e086ed0917fd055b14f63099c2342eb6b86a
```

Scope

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

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	4
Total	11

Findings

High

[H-1] Lack of slippage protection in `TswapPool::swapExactOutput` causes user 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 fluctuate prior to the execution of the transaction, the user may experience significantly less favorable swap rates.

Proof of Concept:

Below is an example of this as a test that can be added to the testsuite:

```
1     function testExactOutputWithSlippage() public {
2        vm.startPrank(user);
3
4        weth.approve(address(pool), type(uint256).max);
5        poolToken.approve(address(pool), type(uint256).max);
6
7        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
8
9        uint256 initialInputReserves = weth.balanceOf(address(pool));
10       uint256 initialOutputReserves = poolToken.balanceOf(address(pool)
             );
11       uint256 initialInputAmount =
12           pool.getInputAmountBasedOnOutput(20e18, initialInputReserves,
                 initialOutputReserves);
13       console.log("Initial Weth reserves in ETH:", initialInputReserves
             / 1e18);
14       console.log("Initial Pool Token reserves in ETH:",
             initialOutputReserves / 1e18);
15       console.log("Initial input in ETH:", initialInputAmount / 1e18);
16
17       pool.swapExactOutput(weth, poolToken, 20e18, uint64(block.
             timestamp));
18
19       uint256 finalInputReserves = weth.balanceOf(address(pool));
20       uint256 finalOutputReserves = poolToken.balanceOf(address(pool));
21       uint256 finalInputAmount = pool.getInputAmountBasedOnOutput(20e18
             , finalInputReserves, finalOutputReserves);
22       console.log("Final Weth reserves in ETH:", finalInputReserves / 1
             e18);
23       console.log("Final Pool Token reserves in ETH:",
             finalOutputReserves / 1e18);
```

```
24          console.log("Input after slippage in ETH:", finalInputAmount / 1
               e18);
25          console.log("Difference in input amount in ETH:", (
               finalInputAmount - initialInputAmount) / 1e18);
26
27          vm.stopPrank();
28          assert(finalInputAmount > initialInputAmount);
29      }
```

This outputs:

```
1
2  Logs:
3  Initial Weth reserves in ETH: 100
4  Initial Pool Token reserves in ETH: 100
5  Initial input in ETH (for a user that wants to swap 20 ETH worth of
      poolToken): 250
6  Final Weth reserves in ETH: 350
7  Final Pool Token reserves in ETH: 80
8  Input in ETH (for a user that wants to swap 20 ETH worth of poolToken)
      after slippage of previous transaction: 1172
9  Difference in input amount in ETH: 921
```

Explanation of Test Results:

- **Initial Reserves**: The test begins with equal reserves for both WETH and PoolToken, each at 100 ETH. This setup reflects a balanced state within the liquidity pool.
- **Initial Input Requirement**: Before the swap, the calculated input amount needed to achieve an exact output of 20 ETH in PoolToken is determined to be 250 ETH worth of WETH.
- **Post-Swap Reserves**: Following the swap, the WETH reserves in the pool increase to 350 ETH. Conversely, the PoolToken reserves decrease to 80 ETH, reflecting the output of the transaction.
- **Slippage Impact**: The significant rise in the input requirement post-swap, to 1172 ETH of WETH, highlights a drastic slippage effect. This increase, from the initial 250 ETH input requirement, showcases the severe impact of market movements on the swap execution.

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();
```

```
10 +         }
11           _swap(inputToken, inputAmount, outputToken, outputAmount);s
```

[H-2] Incorrect Fee Calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees

Description: The `getInputAmountBasedOnOutput` function in the `TSwapPool` contract uses an incorrect multiplier for the fee calculation. The function uses `10000` instead of `1000` , leading to inconsistencies in fee application.

Impact: Protocol takes more fees than expected from users

Recommended Mitigation:

```
1 -     return ((inputReserves * outputAmount) * 10000) / ((outputReserves
      - outputAmount) * 997);
2 +     return ((inputReserves * outputAmount) * 1000) / ((outputReserves
     - outputAmount) * 997);
```

For the optimal solution to maintain consistency in fee-related calculations across the contract, please refer to informational [I-4].

[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: This would also require changing the sellPoolTokens function to accept a new parameter (i.e. minWethToReceive to be passed to swapExactInput)

```
1     function sellPoolTokens(
2         uint256 poolTokenAmount,
3 +       uint256 minWethToReceive,
4         ) external returns (uint256 wethAmount) {
```

```
5 -         return swapExactOutput(i_poolToken, i_wethToken,
      poolTokenAmount, uint64(block.timestamp));
6 +         return swapExactInput(i_poolToken, poolTokenAmount,
      i_wethToken, minWethToReceive, uint64(block.timestamp));
7       }
```

[H-4] Disruption of Protocol Invariant in `TSwapPool::_swap` Due to Extra Token Distribution

Description:
In the `TSwapPool` smart contract, the protocol maintains a strict invariant defined by the equation `x * y = k`. Here: - `x` represents the balance of the pool token, - `y` represents the balance of WETH (Wrapped Ether), - `k` is the constant product of these two balances.

This invariant ensures that the ratio between the token balances remains constant. However, an issue arises due to an additional token distribution mechanism embedded within the `_swap` function. This mechanism disrupts the invariant by altering the balance, leading to a gradual depletion of the pool's resources.

The problematic code is highlighted below:

```
1 swap_count++;
2 if (swap_count >= SWAP_COUNT_MAX) {
3     swap_count = 0;
4     outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
5 }
```

Impact:
The additional token rewards issued by the `_swap` function can be exploited by users performing frequent swaps, leading to a significant drain of the protocol's assets. **Proof of Concept:**
Place the following test case into `TSwapPool.t.sol`:

```
1   function testInvarientBroken() 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        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
              timestamp));
13        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
              timestamp));
```

```
14          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
15          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
16          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
17          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
18          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
19          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
20          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
21
22          int256 startingY = int256(weth.balanceOf(address(pool)));
23          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
24
25          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
26
27          vm.stopPrank();
28
29          uint256 endingY = weth.balanceOf(address(pool));
30
31          int256 actualDeltaY = int256(endingY) - int256(startingY);
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

Recommended Mitigation:

Eliminate the additional 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) {
4 -             swap_count = 0;
5 -             outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
6 -         }
```

Medium

[M-1] `TswapPool::deposit` is missing deadline check allowing post deadline transaction execution

Description: The `deposit` function in the `TswapPool` contract accepts a `deadline` parameter intended to specify the latest timestamp by which the transaction must be executed to ensure that the transaction is processed under acceptable market conditions. However, the function currently lacks the necessary logic to check if the current block timestamp exceeds this deadline. This omission allows transactions to be executed even after the deadline has passed, potentially subjecting liquidity providers to unfavorable changes in market conditions.

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)
    {
```

Low

[L-1] `TswapPool::LiquidityAdded` even has parameters out of order

Description: When the `LiquidityAdded` event is emitted in the `TswapPool::_addLiquidityMintAndTrans` 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

Proof of Concept & 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, whule it declares the named return value `output` it is never assigned a value, nor uses an explict return statement.

Impact: The return value of the `swapExactInput` function will always be zero, providing the caller with incorrect information.

Proof of Concept:

```
1          uint256 inputReserves = inputToken.balanceOf(address(this));
2          uint256 outputReserves = outputToken.balanceOf(address(this));
3
4  -       uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
       inputReserves, outputReserves);
5  +       uint256 output = getOutputAmountBasedOnInput(inputAmount,
   inputReserves, outputReserves);
6
7  -        if (outputAmount < minOutputAmount) {
8  -            revert TSwapPool__OutputTooLow(outputAmount,
   minOutputAmount);
9  +        if (output < minOutputAmount) {
10 +            revert TSwapPool__OutputTooLow(outputAmount,
   minOutputAmount);
11         }
12         }
13
14 -        _swap(inputToken, inputAmount, outputToken, outputAmount);
15 +        _swap(inputToken, inputAmount, outputToken, output);
16     }
```

Informationals

[I-1] Error `PoolFactory::PoolFactory__PoolDoesNotExist` is unutilized and should be removed

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

[I-2] Lacking zero address checks in constructor

Description: The constructor in `PoolFactory` sets the `i_wethToken` state variable based on the input parameter `wethToken` without validating that the input is not the zero address.

To fix this:

```
  constructor(address wethToken) {
+    if(wethToken == address(0)){
        revert();
    }
        i_wethToken = wethToken;
    }
```

[I-3] Incorrect Token Symbol Retrieval in Pool Creation in `PoolFactory::createPool`

Description: In the `PoolFactory` contract, the `createPool` function incorrectly constructs a liquidity token symbol using the name() method of the ERC20 token at the provided tokenAddress, instead of using the symbol() method. This approach can lead to confusion and inconsistency in the expected token symbol format.

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

[I-4] Use of Hard-Coded Values in Contract Calculations

Description: The `TSwapPool` smart contract employs hard-coded values in its calculations. This use of literal constants without named references can decrease code clarity and complicate maintenance or future updates.

Recommended Mitigation:

Introduce named constants at the start of the contract to replace hard-coded values, improving clarity and reducing errors:

```
+ uint256 private constant FEE_MULTIPLIER = 997;
+ uint256 private constant FEE_DENOMINATOR = 1000;
+ uint256 private constant ETHER_UNITS_IN_WEI = 1e18;
```

Utilize these constants in the contract's calculations to ensure consistency and reduce maintenance complexity:

```
return ((inputReserves * outputAmount) * FEE_DENOMINATOR) / ((
    outputReserves - outputAmount) * FEE_MULTIPLIER);
```