TSwap Audit Report

Version 1.0

Cyfrin.io

October 13, 2024

TSwap Audit Report

Dheeraj K

Oct 13, 2024

Prepared by: Dheeraj Lead Auditors:

- Dheeraj Kumar

Table of Contents

Protocol Summary

TSwap is a protocol similar to Uniswap that facilitates exchange of ERC20 tokens for some other tokens or for WETH. This DEX enables swapping of tokens in a permissionless manner. It uses liquidity pools to swap tokens from and to into. Users who deposit their funds into the pool and make swapping actually happen are called liquidity providers. They get share of the swapping fees for providing liquidity.

Disclaimer

Dheeraj & 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.

Audit Details

- 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.

Executive Summary

Issues found

Severity	Number of issues found
High	5
Medium	0
Low	2
Gas	0
Informational	3
Total	10

Findings

High

[H-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, operations that add liquidity to the pool might be executed at unexpected times.

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, // Check for zero amount
        uint64 deadline // @audit-info unused param
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
```

[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 is intended to calculate the amount of tokens a user should deposit given an amount 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.

Impact: Protocol takes more fees than expected from users.

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-3] 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`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact: If market conditions change before the transaction 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 calls `swapExactOutput` for swapping some USDC for 1 WETH.
    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline = some deadline
3. The function has no maxLimit on inputAmount
4. As the transaction is pending in the mempool, the market changes! And the change in price of WETH is HUGE!
5. The user would end up paying more USDC than expected.

Recommended Mitigation: The `swapExactOutput` function should include a `maxInputAmount` so the user has a max cap on how much he's willing to spend for a desired amount of output token.

```
    function swapExactOutput(
        IERC20 inputToken,
+       uint256 maxInputAmount,
.
.
.
        inputAmount = getInputAmountBasedOnOutput(outputAmount,
            inputReserves,outputReserves);
+       if (inputAmount > maxInputAmount) {
+           revert();
+       }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
```

[H-4] `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 specify the amount of pool tokens they're willing to sell in the `poolTokenAmount` parameter. However, the function currently miscalculates the swapped amount.

This is due to the fact that `swapExactOutput` function is called instead of `swapExactInput`. Because users specify the exact amount of input tokens they are willing to spend, not output tokens.

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

Proof of Concept:

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 i.e. `minWethToReceive` to be passed to `swapExactInput`.

```diff
    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.

[H-5] 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 the pool token
- `y`: The balance of WETH
- `k`: The constant product of the two balances

This means, that the ratio between the two amounts should remain constant even if the balance of the two tokens in the protocol changes. Hence, the `k` should also remain constant. However, the extra incentive in the `_swap` function after every 10 swaps breaks this invariant. That means the protocol funds will be drained over time.

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 by the protocol.

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 until all the protocol funds are drained

Proof of Code

Place the following test in `TSwapPool.t.sol`.

```
        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;

            vm.startPrank(user);
            poolToken.mint(user, 100e18);
            poolToken.approve(address(pool), type(uint256).max);
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));

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

            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
            vm.stopPrank();

            uint256 endingY = weth.balanceOf(address(pool));
            int256 actualDeltaY = int256(endingY) - int256(startingY);

            assertEq(expectedDeltaY, actualDeltaY);
        }
```

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, the tokens should be set aside in the similar manner as done with the fees.

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

Medium

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 event is described as below:

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

But `poolTokensDeposited` is logged at second position and `wethDeposited` at third position.

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 `output` it is never assigned, nor any explicit return statement is called.

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

Recommended Mitigation:

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

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

L-3: Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

4 Found Instances

- Found in src/TSwapPool.sol Line: 276

```
            uint256 inputAmountMinusFee = inputAmount * 997;
```

- Found in src/TSwapPool.sol Line: 295

```
                ((outputReserves - outputAmount) * 997);
```

- Found in src/TSwapPool.sol Line: 455

```
                  1e18,
```

- Found in src/TSwapPool.sol Line: 464

```
                  1e18,
```

Informational

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

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

[I-2] Lacks zero address checks

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

[I-3] `PoolFactory::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());
```