TSwap Protocol Audit Report

Version 1.0

Panayot.io

January 13, 2024

Protocol Audit Report

Panayot Kostov

January 13, 2024

Prepared by: Panayot Lead Auditors: - xxxxxxx

Table of Contents

  * [H-5] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`
    - 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
    - Informationals
      * [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] Event is missing `indexed` fields

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The YOUR_NAME_HERE 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

	Impact			
	Low	M	M/L	L

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

Audit Details

Scope

Roles

Executive Summary

Issues found

Severtity	Number of issues found
High	4
Medium	2
Low	2
Info	9
Total	17

Findings

High

WRITE POC for ALL HIGHS ### [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, 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.

```
 1  function deposit(
 2          uint256 wethToDeposit,
 3          uint256 minimumLiquidityTokensToMint, // LP tokens -> if empty,
                    we can pick 100% (100% == 17 tokens)
 4          uint256 maximumPoolTokensToDeposit,
 5          uint64 deadline
 6      )
 7          external
 8  +       revertIfDeadlinePassed(deadline)
 9          revertIfZero(wethToDeposit)
10          returns (uint256 liquidityTokensToMint)
11      {
```

[H-2] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocll 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.

Impact: Protocol takes more fees than expected from users.

Proof of Concept: 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.

To test this, include the following code in the `TSwapPool.t.sol` file:

```
 1  function testFlawedSwapExactOutput() public {
```

```
2        uint256 initialLiquidity = 100e18;
3        vm.startPrank(liquidityProvider);
4        weth.approve(address(pool), initialLiquidity);
5        poolToken.approve(address(pool), initialLiquidity);
6
7        pool.deposit({
8            wethToDeposit: initialLiquidity,
9            minimumLiquidityTokensToMint: 0,
10           maximumPoolTokensToDeposit: initialLiquidity,
11           deadline: uint64(block.timestamp)
12       });
13       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-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`, 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();
10  +         }
11            _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 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.

Proof of Concept:

1. Run the following test in TSwapPool.t.sol - the pool token that the user holds is much lower than expected after executing sellPoolTokens.

```
 1  function testSellPoolTokensPOC() 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
 7          vm.startPrank(user);
 8          weth.approve(address(pool), 10e18);
 9
10          uint256 initialUserPoolTokenBalance = poolToken.balanceOf(user)
                ;
11          console.log("Initial pool token ", initialUserPoolTokenBalance)
                ;
12          uint256 initialUserWethBalance = weth.balanceOf(user);
13          console.log("Intial weth balance ", initialUserWethBalance);
14
15          uint256 poolTokenToSell = 1e18;
16          poolToken.approve(address(pool), 15e18);
17          pool.sellPoolTokens(poolTokenToSell);
18
19          uint256 afterlUserPoolTokenBalance = poolToken.balanceOf(user);
20          console.log("After pool token ", afterlUserPoolTokenBalance);
```

```
21          uint256 afterlUserWethBalance = weth.balanceOf(user);
22          console.log("After weth token ", afterlUserWethBalance);
23
24          assertEq(afterlUserPoolTokenBalance, 15e18 - poolTokenToSell);
25          assert(afterlUserWethBalance > initialUserWethBalance);
26          assertEq(afterlUserWethBalance, 15987158034397061298);
27
28          //   After pool token   4868595686048043120
29          //   After weth token  16000000000000000000
30          //   Error: a == b not satisfied [uint]
31          //          Left:  4868595686048043120
32          //         Right: 14000000000000000000
33          //   Error: a == b not satisfied [uint]
34          //          Left: 16000000000000000000
35          //         Right: 15987158034397061298
36      }
```

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`)

```
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      }
```

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

[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 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 follow block of code is responsible for the issue.

```
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: 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         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         pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
22
```

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

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

Low

[L-1] `TSwapPool::LiquidityAdded` event 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.

Recommended Mitigation:

```
1  - emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
2  + 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.

Proof of Concept:

1. Run the test bellow.

```
1      function testSwapExactInputPOC() 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          vm.startPrank(user);
9          poolToken.approve(address(pool), 10e18);
10         // After we swap, there will be ~110 tokenA, and ~91 WETH
11         // 100 * 100 = 10,000
12         // 110 * ~91 = 10,000
13         uint256 expected = 9e18;
14
15         uint256 output = pool.swapExactInput(poolToken, 10e18, weth,
               expected, uint64(block.timestamp));
16         assertEq(output, expected);
17     }
```

Recommended Mitigation:

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

Informationals

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

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

[I-2] Lacking zero address checks

```
1     constructor(address wethToken) {
2 +       if(wethToken == address(0)) {
3 +           revert();
4 +        }
5       i_wethToken = wethToken;
6     }
```

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

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

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

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.

- Found in src/TSwapPool.sol: Line: 44
- Found in src/PoolFactory.sol: Line: 37
- Found in src/TSwapPool.sol: Line: 46
- Found in src/TSwapPool.sol: Line: 43