TSwap Initial Audit Report

Version 0.1

Cyfrin.io

May 2, 2024

TSwap Audit Report

Kostiantyn Osadchii

May 2, 2024

TSwap Audit Report

Prepared by: Kostiantyn Osadchii Lead Auditors:

- Kostiantyn Osadchii

Assisting Auditors:

- None

Table of contents

See table

About Kostiantyn Osadchii

Enthusiastic and detail-oriented Junior Solidity Developer with a strong foundation in blockchain technology and smart contract development. Proficient in writing secure and efficient code using Solidity for decentralized applications. Excited to contribute innovative solutions and leverage emerging technologies to drive success in a dynamic development environment

Disclaimer

The Kostiantyn Osadchii 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
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

Audit Details

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

```
1ec3c30253423eb4199827f59cf564cc575b46db
```

Scope

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

Protocol Summary

TSWAP is an constant-product AMM that allows users permissionlessly trade WETH and any other ERC20 token set during deployment. Users can trade without restrictions, just paying a tiny fee in each

swapping operation. Fees are earned by liquidity providers, who can deposit and withdraw liquidity at any time.

Roles

- Liquidity Provider: An account who deposits assets into the pool to earn trading fees.
- User: An account who swaps tokens.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	4
Gas	0
Total	11

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` function causes protocol to take too many tokens from users.

Description: The `getInputAmountBasedOnOutput` function is designed to calculate the input amount required to achieve a specified output amount. This function is crucial for determining the amount of input tokens needed to swap for a certain amount of output tokens in a liquidity pool. However, the calculation of the fee within this function is incorrect due to an oversight(10000 instead of 1000) in the fee calculation formula.

Impact: The incorrect fee calculation can lead to several issues:

- Inaccurate Token Swap: The calculated input amount might not accurately reflect the actual amount needed to achieve the desired output amount, leading to inaccurate token swaps.
- Financial Losses: Users might end up paying more than expected for their token swaps, leading to financial losses.
- Trust and Reputation: Incorrect fee calculations can erode trust in the liquidity pool, potentially affecting its reputation and usage.

Proof of Concept: Consider a scenario where a user wants to swap a certain amount of input tokens for output tokens. If the fee calculation is incorrect, the user might end up paying more than the expected fee, leading to a discrepancy between the expected and actual output amount.

PoC

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

```
function testCollectsTooBigFeesOnSwapExactOutput() public {
        uint256 startingWeth = 100e18;
        uint256 startingPoolToken = 100e18;
        uint256 outputAmount = 10e18;
        uint256 epectedInputAmount = pool.getInputAmountBasedOnOutput(
            outputAmount,
            startingPoolToken,
            startingWeth
        );
        poolToken.mint(user, epectedInputAmount);

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), startingWeth);
        poolToken.approve(address(pool), startingPoolToken);
        pool.deposit(
            startingWeth,
            startingWeth,
            startingPoolToken,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        vm.startPrank(user);
        poolToken.approve(address(pool), epectedInputAmount);
        uint256 userPoolTokenBalanceBefore = poolToken.balanceOf(user);
        pool.swapExactOutput(
            poolToken,
            weth,
            outputAmount,
            uint64(block.timestamp)
        );
        uint256 userPoolTokenBalanceAfter = poolToken.balanceOf(user);
        vm.stopPrank();

```

```
35          //correct calculation
36          uint256 expectedInputAmountWithCorrectFee = ((startingPoolToken
                *
37              outputAmount) * 1000) / ((startingWeth - outputAmount) *
                    997);
38
39          uint256 actualInputAmount = userPoolTokenBalanceBefore -
40              userPoolTokenBalanceAfter;
41          console.log(
42              "Input amount with correct fee calculation",
43              expectedInputAmountWithCorrectFee
44          );
45          console.log(
46              "Actual input amount with wrong fee calculation",
47              actualInputAmount
48          );
49          assert(actualInputAmount != expectedInputAmountWithCorrectFee);
50      }
```

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
13 -           ((inputReserves * outputAmount) * 10000) /
14 -           ((outputReserves - outputAmount) * 997);
15 +       return
16 +           ((inputReserves * outputAmount) * 1000) /
17 +           ((outputReserves - outputAmount) * 997);
18     }
```

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

Description: The `swapExactOutput` does not include any sort of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput`, where function specifies a `minOutputAmount`, 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 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 protocol.

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

Proof of Concept:

PoC

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

```
function testSellPoolTokensFunctionUsingWrongSwapMethod() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(50e18, 0, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 userBalanceBefore = poolToken.balanceOf(user);
        vm.startPrank(user);
        uint256 amountToSell = 1e10;
        poolToken.approve(address(pool), type(uint256).max);
        pool.sellPoolTokens(amountToSell);
        vm.stopPrank();
        uint256 userBalanceAfter = poolToken.balanceOf(user);
        console.log(
            "Amount of pool tokens that user wanted to sell: ",
            amountToSell
        );
        console.log(
            "Actual amount of pool tokens that user sold because of
               wrong swap method: ",
            userBalanceBefore - userBalanceAfter
        );
        assert(userBalanceAfter != (userBalanceBefore - amountToSell));
    }
```

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 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.

```
        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.

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`.

```

    function testSwapBreaksInvariant() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(50e18, 0, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 outputWeth = 1e10;
        poolToken.mint(user, 100e18);
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        for (uint256 i = 0; i < 9; i++) {
            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) - startingY;

        assert(actualDeltaY != expectedDeltaY);
    }
```

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.

```
-        swap_count++;
-        // Fee-on-transfer
-        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 transactions to complete even after the deadline

Description: The `deposit` function accepts a `deadline` parameter, which according the documentation id "The deadline for the transaction to be completed by". However, the function does not check if the transaction is being executed after the specified deadline.

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.

```
Warning (5667): Unused function parameter. Remove or comment out the
    variable name to silence this warning.
   --> src/TSwapPool.sol:117:9:
```

```
3       |
4   117 |         uint64 deadline
5       |
```

Recommended Mitigation: To mitigate this issue, the contract should include a check to ensure that the transaction is executed before the specified deadline. This can be achieved by comparing the current block timestamp with the deadline parameter. If the current timestamp is greater than the deadline, the transaction should be reverted.

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

Low

[L-1] Incorrect order of rarameters in `TSwapPool::LiquidityAdded` event emission

Description: The `TSwapPool::_addLiquidityMintAndTransfer` function is responsible for adding liquidity to the pool by minting liquidity tokens and transferring WETH and pool tokens from the user to the contract. It also emits a `LiquidityAdded` event to log the details of the liquidity addition. However, the order of parameters in the `LiquidityAdded` event emission does not match the expected or logical order, which could lead to confusion and potential misinterpretation of the event data.

Impact: The incorrect order of parameters in the `LiquidityAdded` event could have several implications:

- Operational Inconsistency: Operators and users monitoring the contract's events might interpret the event data incorrectly, leading to confusion and potential errors in their analysis or operations.
- Audit and Debugging Complexity: Auditors and developers debugging the contract might overlook the issue due to the misleading event data, increasing the complexity of identifying and fixing bugs.
- User Experience: Users or dApps interacting with the contract might rely on the event data for their operations, potentially leading to incorrect assumptions or decisions based on the

misleading information.

Proof of Concept: Consider a scenario where a user or a dApp listens for the `LiquidityAdded` event to track the liquidity contributions. If the event parameters are emitted in the wrong order, the user or dApp might mistakenly believe that the amount of WETH deposited is actually the amount of pool tokens deposited, and vice versa. This could lead to incorrect calculations or decisions based on the event data.

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 return the actual amount of tokens bought by the caller. However, while it declares the named return value `output` it is never assigned a value, nor uses an explicit return statement.

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

Proof of Concept:

PoC

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

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

        vm.startPrank(user);
        uint256 expected = 9e18;
        poolToken.approve(address(pool), 10e18);
        uint256 swapReturn = pool.swapExactInput(
            poolToken,
            10e18,
            weth,
            expected,
            uint64(block.timestamp)
        );
```

```
        vm.stopPrank();
        assertEq(swapReturn, 0);
    }
```

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);

    }
```

Informational

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is n ot used and can be removed

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

[I-2] Lacking zero address checks

`PoolFactory.sol`

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

TSwapPool.sol

```
    constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
+       if(poolToken == address(0) || wethToken == address(0)) {
+           revert();
+       }
        i_wethToken = IERC20(wethToken);
        i_poolToken = IERC20(poolToken);
    }
```

[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());
```

[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/PoolFactory.sol Line: 35

```
    event PoolCreated(address tokenAddress, address poolAddress);
```

- Found in src/TSwapPool.sol Line: 52

```
    event LiquidityAdded(
```

- Found in src/TSwapPool.sol Line: 57

```
1       event LiquidityRemoved(
```

- Found in src/TSwapPool.sol Line: 62

```
1       event Swap(
```