TSwap Protocol Audit Report

Version 1.0

Cyfrin.io

July 19, 2024

TSwap Protocol Audit Report

Nengak Emmanuel Goltong

July 19, 2024

Prepared by: Spomaria Lead Auditors: - Nengak Emmanuel Goltong

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees
    * [H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive fewer tokens
    * [H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens
    * [H-4] In `TWswapPool::_swap` the extra tokens given to users after every `swapCount` breaks the the protocol invariant of `x * y = k`.

- Medium
  - [M-1] `TSwapPool::deposit` function is missing the deadline check causing transactions to complete even after the deadline.
- Lows
  - [L-1] `TSwapPool::LiquidityAdded` event has parameters out of order causing event to emit incorrect information
  - [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 checks
  - [I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`

Protocol Summary

Protocol does X, Y, Z

Disclaimer

The Spomaria 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

Severity	Number of Issues Found
High	4
Medium	1
Low	2
Gas	3
Info	0
Total	10

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 output tokens. However, the function currently miscalculates the amount. When calculating the fees, the function scales the amount of 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Proof of Concept:

Recommended Mitigation:

```
function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {

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

    }
```

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

Description: The `swapExactOutput` function does not include any form of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput`, where the function specifies a `minOutputAmount`, the `TSwapPool::swapExactOutput` 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 `maxInputAmount` 4. As the transaction is pending in the mempool, the market changes! And the price moves HUGH -> 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 specified amount, and can predict how much they will spend on the protocol.

```
function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxInputAmount,
        uint64 deadline
    )
        public
        revertIfZero(outputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 inputAmount)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));
        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 easily sell their poolTokens and receive WETH in exchange. Users indicate how much poolTokens they are 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:

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`)

```
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 may be wise to add a deadline to the function, as there is currently no deadline.

[H-4] In `TWswapPool::_swap` the extra tokens given to users after every `swapCount` breaks the 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 the WETH - `k`: the constant product of the two tokens

This means that whenever the balances of the tokens change in the protocol, the ratio between the two amounts should remain the constant, hence the `k`. However, this is broken due to the extra inceptive 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)
            ;
    }
```

Impact: A user could maliciously drain the protocol of funds by performing a lot of swaps and collecting the extra inceptive given out by the protocol.

Simply put, the protocol's core invariant is broken.

Proof of Concept: 1. A user swaps ten times and collects an extra incentive of `1_000_000_000_000_000_000` tokens 2. That user just continues to swap until all the protocol funds are drained.

Proof of Code Place the following into `TSwapPool.t.sol`.

```
function testInvariantIsBroken() 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.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
        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));
        vm.stopPrank();


        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);
        vm.prank(user);
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp));

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

        assertEq(expectedDeltaY, actualDeltaY);
    }
```

Recommended Mitigation: Remove the extra inceptive mechanism. If you want to keep this in, we should account for this change in the `x * y = k` protocol invariant. Or, we should set aside tokens in the same way we do with fees.

```
-    swap_count++;
-    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` function is missing the deadline check causing transactions to complete even after the deadline.

Description: The `TSwapPool::deposit` function accepts a 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, operators that add liquidity to the pool might be executed at unexpected times, in the market conditions where the deposit rate is unfavorable.

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

Proof of Concept: the `deadline` parameter is unused

Recommended Mitigation: Consider making the following changes to the function

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

Lows

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order causing event to emit incorrect information

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran`
, it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the third parameter position, whereas the `wethTokenToDeposit` value should go in the second parameter 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 a value nor uses an explicit return statement.

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

Proof of Concept:

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(inputAmount, inputReserves, outputReserves);

    if (outputAmount < minOutputAmount) {
        revert TSwapPool__OutputTooLow(outputAmount, minOutputAmount);
    }


    _swap(inputToken, inputAmount, outputToken, outputAmount);

}
```

Informationals

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

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

[I-2] lacking zero 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());
```