TSwap Protocol Audit Report

Version 1.0

Cyfrin.io

March 5, 2024

TSwap Audit Report

IzuMan

March 4, 2024

Prepared by: IzuMan Lead Auditors:

- xxxxxxx

Table of Contents

- [H-4] In `TSwapPool:_swap` The extra tokens given to users after every `swapCount` breaks the protocol's invariance
- Medium
  * [M-1] `TSwapPool:deposit` is missing the deadline check causing the transaction to complete after the deadline
  * [M-2] Weird ERC20 tokens can break this protocol. For, example rebasing, fee on transfer, or tokens with fallback functions etc.
- Low
  * [L-1] `TSwapPool:LiquidityAdded` event has parameters out of order.
  * [L-2] `TSwapPool:swapExactInput` default value return is incorrect.
- Informational
  * [I-1] `PoolFactory:PoolFactory__PoolDoesNotExist` is not used and should be removed
  * [I-2] `PoolFactory:constructor` Does not check for zero address
  * [I-3] `PoolFactory:createPool` should use `.symbol()` instead of `.name()`
  * [I-4] `TSwapPool:constructor` Does not check for zero address
  * [I-5] `TswapPool:deposit` the variable `MINIMUM_WETH_LIQUIDITY` does not need to be included in revert statement since it is a constant.
  * [I-6] This `TswapPool:poolTokenReserves` is not used and can be removed to save gas
  * [I-7] `TSwapPool:deposit` does not follow best practice of CEI
  * [I-8] Remove magic numbers in the contract. "Magic numbers" make the code harder to understand and is prone to making future mistakes.
  * [I-9] The `TSwapPool:swapExactInput` is missing NatpSpec.
  * [I-10] The `TSwapPool:swapExactInput` should be changed to external to save gas since it is never called by `TSwapPool`.
  * [I-11] The `TSwapPool:totalLiquidityTokenSupply` is not used by the contract and should be marked external.

Protocol Summary

This project is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX). T-Swap is known as an Automated Market Maker (AMM) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset. It is similar to Uniswap. To understand Uniswap, please watch this video: Uniswap Explained

Disclaimer

The IzuMan 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

Scope

–*src/PoolFactory.sol* –src/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

Many issues were found in the protocol and all high impact issue should be address before this protocol is deployed.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	11
Total	19

Findings

[H-1] Incorrect Fee calculation in `TSwapPool:getInputAmountBasedOnOutput` causes the protocol to take too many coins from the user.

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user shoudl deposit given an amout of output tokens. However, currently the function scales the amount of fees by 10_000 instead of 1_000.

Impact: The user will be charged high and incorrect fees by the protocol

Proof of Concept:

Recommended Mitigation:

```
1 -       return ((inputReserves * outputAmount) * 10000) / ((
    outputReserves - outputAmount) * 997);
2 +       return ((inputReserves * outputAmount) * 1_000) / ((
    outputReserves - outputAmount) * 997);
```

[H-2] `TSwapPool:swapExactOutput` does not have slippage protection. User could receive a much less amount of coins than expected.

Description: `swapExactOutput` does not have any slippage protection. This function is similar to `TSwapPool:swapExactInput` where the function specifies a `minOutputAmount`, in this situation we would want a parameter to specify the `maxInputValue` to protect the user from slippage.

Impact: Users could get a much worse swap than expected.

Proof of Concept:

1. The price of 1 WETH right now is $1000 USDC.
2. User inputs `swapExactOutput` expecting the price per WETH to be ~$1,000.
    1. Enter the `outputAmount` as 1 WETH
3. Transaction is submitted and sits in the mem-pool.
4. The price of WETH changes to 1 WETH or $10,000 USDC.
5. User transaction finally goes through and $10,000 is transferred from the user and they receive 1 WETH.

PoC

Recommended Mitigation: We should include a `maxInputAmount` paramter to the function input data to set the max amount they will spend on th transaction.

```
1
2   function swapExactOutput(
3           IERC20 inputToken,
4           IERC20 outputToken,
5           uint256 outputAmount,
6   +       uint256 maxInputAmount,
7           uint64 deadline
8       )
9           public
10          revertIfZero(outputAmount)
11          revertIfDeadlinePassed(deadline)
12          returns (uint256 inputAmount)
13      {
14          uint256 inputReserves = inputToken.balanceOf(address(this));
15          uint256 outputReserves = outputToken.balanceOf(address(this));
16
17          inputAmount = getInputAmountBasedOnOutput(outputAmount,
               inputReserves, outputReserves);
18
19  +        if(inputAmount > maxInputAmount){
20  +            revert();
21  +        }
```

```
22
23          _swap(inputToken, inputAmount, outputToken, outputAmount);
24      }
```

[H-3] `TSwapPool:sellPoolTokens` mismatches the input and output cause users to receive the incorrect amount of tokens.

Description: The `sellPoolTokens` function is intended for user to swap between poolTokens and WETH tokens. Users indicate how many pool tokens they are willing to sell. However, the function misscalculates the amount being swapped.

This is due to the fact that the `swapExactOutput` function is called where as the `exactExactInput` 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 the protocols functionality.

Proof of Concept:

PoC

```
1  function test_sellPoolTokensSwapsIncorrectAmountOfTokens() public {
2          uint256 poolTokensToSell = 10 ether;
3          uint256 tokensToMint = 200 ether;
4          //setting up pool Liquidity
5          vm.startPrank(liquidityProvider);
6          weth.approve(address(pool), tokensToMint);
7          poolToken.approve(address(pool), tokensToMint);
8          pool.deposit(tokensToMint, tokensToMint, tokensToMint, uint64(
               block.timestamp));
9          vm.stopPrank();
10
11         vm.startPrank(user);
12         poolToken.mint(address(user), tokensToMint);
13         uint256 startingPoolTokensBalance = poolToken.balanceOf(user);
14         poolToken.approve(address(pool), tokensToMint);
15         weth.approve(address(pool), tokensToMint);
16         pool.sellPoolTokens(poolTokensToSell);
17         uint256 endingPoolTokensBalance = poolToken.balanceOf(user);
18         int256 differenceInBalance = int256(startingPoolTokensBalance)
               - int256(endingPoolTokensBalance);
19         if (differenceInBalance < 0) {
20             differenceInBalance = differenceInBalance * (-1);
21             console.log("The user incorrectly bought pool tokens!!",
                   uint256(differenceInBalance));
22             return;
23         } else {
```

```
24              console.log("The user sold poolTokens", uint256(
                    differenceInBalance));
25              assertEq(uint256(differenceInBalance), poolTokensToSell);
26          }
27      }
```

Recommended Mitigation: Use `swapExactInput` instead.

```
1  function sellPoolTokens(
2      uint256 poolTokenAmount,
3  +   uint256 minWethToReceive
4
5  ) external returns (uint256 wethAmount) {
6
7  -         return swapExactOutput(i_poolToken, i_wethToken,
       poolTokenAmount, uint64(block.timestamp));
8  +         return swapExactInput(i_poolToken,poolTokenAmount, i_wethToken
       , minWethToReceive, uint64(block.timestamp));
9      }
```

Additionally, it might be wise to add a deadline to the function as there is no deadline.

[H-4] In `TSwapPool:_swap` The extra tokens given to users after every `swapCount` breaks the protocol's invariance

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 of the 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 broken due to the protocol giving out incentive tokens in the `_swap` function everytime a user reaches the `swapCount`. This means that over time an attacker can slowly drain funds from the protocol by receiving trading incentives.

Impact: Overtime the protocol funds could be drained by a user collecting all the trading incentives. The protocol's invariant is broken.

The following block of code is the coause for a broken invariant

```
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  }
```

Proof of Concept:

1. User swaps 10 times and collects the incentive of `1_000_000_000_000_000_000`.
2. User swap 10 times and collects the incentive.
3. Repeats and repeat until the protocol funds are drained.

PoC

Run the following test in Foundry:

```
1
2   function test_breaksProtocolInvariance() public {
3           vm.startPrank(liquidityProvider);
4           weth.approve(address(pool), 100e18);
5           poolToken.approve(address(pool), 100e18);
6           pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
7           vm.stopPrank();
8
9           uint256 outputWeth = 1e18;
10          int256 startingY = int256(weth.balanceOf(address(pool)));
11          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
12          vm.startPrank(user);
13          poolToken.mint(user, 10000 ether);
14          poolToken.approve(address(pool), type(uint256).max);
15          for (uint256 i; i < 10; i = i + 1) {
16              pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                    block.timestamp));
17          }
18          vm.stopPrank();
19
20          int256 endingY = int256(weth.balanceOf(address(pool)));
21          int256 actualDeltaY = int256(endingY) - int256(startingY);
22
23          assertEq(expectedDeltaY, actualDeltaY);
24      }
```

Recommended Mitigation: Remove the incentives. If you want to keep the incentives you should set aside tokens like fees for incentives.

Recommeded Mitigation

```
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 -         }
```

Medium

[M-1] `TSwapPool:deposit` is missing the deadline check causing the transaction to complete after the deadline

Description: The `deposit` function accepts a dealine parameter which according to the documentation is "/// @param deadline the deadline for the transaction to be completed by". Howver, the parameter is not used and it could be executed a unfavorable market conditions.

Impact: Transactions could be later executed at unexpected times. This also makes the transaction vulnerable to MEV attacks.

Proof of Concept:

Recommended Mitigation: The recommended code changes can be seen below:

Recommended Code Change

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        // This is not being used
        // @audit-high This will cause "stale" transactions to go through since the deadline is ignored
        // Likelihood: high
        uint64 deadline
    )
+       revertIfDeadlinePassed(deadline)
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
```

[M-2] Weird ERC20 tokens can break this protocol. For, example rebasing, fee on transfer, or tokens with fallback functions etc.

Description Malicious/weird ERC20 tokens can break the invariance of the protocol.

Impact It may be possible for all the fund of the protocol to be stolen

Proof of Concept

Mitigation Carefully audit any ERC20 token that is placed into a pool.

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

Impact: Event emission is incorrect leading to off chain incorrect analysis

Recommended Mitigation:

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

[L-2] `TSwapPool:swapExactInput` default value return is incorrect.

Description: The `swapExactInput` function is expected to return teh actual amount fo atokens bought by the caller. However, while it decales the return value it is assigned a value and will always return zero and does not have an explicit return statement.

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

Proof of Concept:

PoC

```
1
2  function test_swapExactInputAlwasReturnsZero() public {
3          uint256 poolTokenInputAmount = 1 ether;
4          uint256 wethMinOutputAmount = 0.5 ether;
5
6          vm.startPrank(liquidityProvider);
7          weth.approve(address(pool), 100e18);
8          poolToken.approve(address(pool), 100e18);
9          pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
10         vm.stopPrank();
11
12         vm.startPrank(user);
13         poolToken.approve(address(pool), 100e18);
14         uint256 outputReturn =
15             pool.swapExactInput(poolToken, poolTokenInputAmount, weth,
                wethMinOutputAmount, uint64(block.timestamp));
16
17         assertEq(outputReturn, 0);
18     }
```

Recommended Mitigation: Change the return value to `outputAmount`

```
1
2 returns (
3 -           uint256 output
4 +           uint256 input
5         )
```

Informational

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

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

[I-2] `PoolFactory:constructor` Does not check for zero address

```
1     constructor(address wethToken) {
2             // @audit add a zero address check
3             i_wethToken = wethToken;
4         }
```

[I-3] `PoolFactory: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] `TSwapPool:constructor` Does not check for zero address

```
1 {
2   i_wethToken = IERC20(wethToken);
3   i_poolToken = IERC20(poolToken);
4 }
```

[I-5] `TswapPool:deposit` the variable `MINIMUM_WETH_LIQUIDITY` does not need to be included in revert statement since it is a constant.

[I-6] This `TswapPool:poolTokenReserves` is not used and can be removed to save gas

Description

```
- uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

[I-7] `TSwapPool:deposit` does not follow best practice of CEI

Description

```
else {
            // This will be the "initial" funding of the protocol. We
               are starting from blank here!
            // We just have them send the tokens in, and we mint
               liquidity tokens based on the weth
+            liquidityTokensToMint = wethToDeposit;
            _addLiquidityMintAndTransfer(wethToDeposit,
               maximumPoolTokensToDeposit, wethToDeposit); // this is
               an

-          liquidityTokensToMint = wethToDeposit;
        }
```

[I-8] Remove magic numbers in the contract. “Magic numbers” make the code harder to understand and is prone to making future mistakes.

```
        uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * 1000) +
           inputAmountMinusFee;
```

[I-9] The `TSwapPool:swapExactInput` is missing NatpSpec.

[I-10] The `TSwapPool:swapExactInput` should be changed to external to save gas since it is never called by `TSwapPool`.

[I-11] The `TSwapPool:totalLiquidityTokenSupply` is not used by the contract and should be marked external.