# TSwap Audit Report

Version 1.0

*Boris Kolev*

April 30, 2024

# Protocol Audit Report

Boris Kolev

April 30, 2024

Prepared by: Boris Kolev

Lead Auditors:

- Boris Kolev

## Table of Contents

## 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

Boris Kolev 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 |

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

## Audit Details

### Scope

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda
- In Scope:

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- Tokens:
  - Any ERC20 token

### Actors / 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

*The audit resulted in four `high` findings, one `medium`, one `low`, 1 `gas` and four `informational` findings.*

*The audit took 5 day of reviwing, utilizing `manual review`, `fuzz testing`, etc.*

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 4 |
| Medium | 1 |
| Low | 2 |
| Informational | 4 |
| Gas | 0 |
| Total | 11 |

## Findings

### High

#### [H-1] The `TSwapPool::getInputAmountBasedOnOutput` function has an incorrect calculation for the fees, causing users to pay too much

**Description** The `getInputAmountBasedOnOutput` function calculates the input amount based on the output amount and the reserves. The calculation for the fees is incorrect, causing users to pay too much. When calculating it uses `10000` instead of `1000`.

**Impact** Users pay too much when swapping tokens.

**Proof of concept** This is shown in the `TSwapPool.t.sol::testInvalidFeeOnGetInputAmountBasedOnO` test.

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

        uint256 numerator = ((10 ** 18 * 5 ** 18) * 1000); // How it should be
        uint256 denominator = ((10 ** 18 - 5 ** 18) * 997);
        uint256 expectedResult = numerator / denominator;
        uint256 acutalResult = pool.getInputAmountBasedOnOutput(5e18, 10e18, 10e18);
        assertNotEq(expectedResult, acutalResult);
}
```

**Recommended mitigation** Change the `10000` to `1000`.

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

#### [H-2] The `TSwapPool::swapExactOutput` has no slippage protection, which causes users to potentially receive less than expected

**Description** The `swapExactOutput` function does not have slippage protection. This function is similar to what is done in `swapExactInput`, where the function specifies `minOutputAmount` to protect against slippage. This is why the `swapExactOutput` function should also have a `maxInputAmount` parameter to protect against slippage.

**Impact** If the market conditions change, users could receive less than expected.

**Proof of concept**

1. The price of WETH is 1,000 USDC.
2. User inputs a `swapExactOutput` looking for 1 WETH.

    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline

3. The function does not offer a maxInput amount.
4. As the transaction is pending, the price of WETH increases to 10,000 USDC.
5. The transaction completes, and the user pays 10000 USDC for 1 WETH.

**Recommended mitigation** Add a `maxInputAmount` parameter to the `swapExactOutput` function.

```
inputAmount = getInputAmountBasedOnOutput(outputAmount, inputReserves, outputReserves);
+ if (inputAmount > maxInputAmount) { // maxInputAmount should be added in the function signature
+     revert();
+}
```

### [H-3] The `TSwapPool::sellPoolTokens` mismatches input and output tokens, causing users to receive the incorrect amount of tokens

**Description** The `sellPoolTokens` function swaps pool tokens for WETH. Users indicate how many pool tokens they are willing to sell in the `poolToken` 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 should be called.

**Impact** Users receive the incorrect amount of tokens when selling pool tokens.

**Proof of concept** This is shown in the `TSwapPool.t.sol::testInvalidSellPoolTokens` test.

```
function testInvalidSellPoolTokens() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), type(uint256).max);
        poolToken.approve(address(pool), type(uint256).max);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));

        uint256 outputReserves = weth.balanceOf(address(pool));
        uint256 inputReserves = poolToken.balanceOf(address(pool));

        uint256 userWethBalance = weth.balanceOf(liquidityProvider);

        // how selling PT for WETH works
        uint256 inputAmountMinusFee = 10 ** 18 * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * 1000) +
           inputAmountMinusFee;
        uint256 expectedResult = numerator / denominator;

        // how selling PT for WETH is implemented
        uint256 acutalResult = pool.sellPoolTokens(10 ** 18);
        assertNotEq(expectedResult, acutalResult);
        vm.stopPrank();
}
```

**Recommended mitigation** Change the `swapExactOutput` to `swapExactInput`. Note that this will require changing the `sellPoolTokens` to include `minWethToReceive` as a parameter.

```
-    function sellPoolTokens(
-        uint256 poolTokenAmount
-    ) external returns (uint256 wethAmount) {
-        return
-            swapExactOutput(
-                i_poolToken,
-                i_wethToken,
-                poolTokenAmount,
-                uint64(block.timestamp)
-            );
+    function sellPoolTokens(uint256 poolTokenAmount, uint256
   minWethToReceive) external returns (uint256 wethAmount) {
+        // audit-high need to change poolTokenAmount to weth Amount
+        return swapExactInput(i_poolToken, poolTokenAmount,
   i_wethToken, minWethToReceive, uint64(block.timestamp));
     }
```

### [H-4] The `TSwapPool::_swap` function mints an additional token on a specific number of swaps, which breaks the protocol invariant `x * y = k`

**Description** The protocol follows a strict invariant `x * y = k`, where `x` is the amount of pool tokens and `y` is the amount of WETH. The `swap` function mints an additional token on a specific number of swaps (i.e. 10 swaps), which breaks the invariant, and would eventually lead to the protocol being drained of all its funds.

**Impact** A user could maliciously drain the protocol by executing a large number of swaps, collecting extra incentives. Most simply put, the protocol's core invariant is broken.

**Proof of concept** This is shown in the `TSwapPool.t.sol::testBrokenInvariant` test.

```
function testBrokenInvariant() 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 wethAmount = 1e17;

        vm.startPrank(user);
        weth.mint(user, wethAmount);
        poolToken.mint(user, 10e27);
        weth.approve(address(pool), type(uint256).max);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, wethAmount, uint64(block.timestamp));

        uint256 startingY = weth.balanceOf(address(pool));
        int256 expectedDeltaY = int256(wethAmount) * (-1);

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

        uint256 finalY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(finalY) - int256(startingY);
        assertEq(actualDeltaY, expectedDeltaY);
    }
```

**Recommended mitigation** Remove the minting of additional tokens in the `_swap` function.

```
-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] The `TSwapPool::deposit` is missing `deadline` check, causing transactions to complete even after the deadline

**Description** The `deposit` function accepts a `deadline` parameter, but it is not used in the function. This can lead to transactions completing even after the deadline has passed. This also makes the protocol vulnerable to front-running attacks.

**Impact** Transactions could be sent when market conditions are unfavorable, even when adding a deadline paramater.

**Proof of concept** The `deadline` parameter is not used.

**Recommended mitigation** Add a check to ensure that the transaction is completed before the deadline.

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

## Low

### [L-1] The `TSwapPool::LiquidityAdded` event has parameters out of order

**Description** When the `LiquidityAdded` event is emitted, the parameters are out of order. This can lead to confusion when reading the event logs. The `poolTokensToDeposit` should go in the third position, and the `wethToDeposit` should go in the second position.

**Impact** Event emission is incorrect, leading to off-chain data misinterpretation.

**Proof of concept** The `LiquidityAdded` event has the parameters out of order.

**Recommended mitigation** Change the order of the parameters in the `LiquidityAdded` event.

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

### [L-2] Defaul value returned by `TSwapPoool::swapExactInput` results in incorrect return value

**Description** The `swapExactInput` function returns a default value which is not used anywhere in the code.

**Impact** The default value is not used and can lead to confusion.

**Recommended mitigation** Remove the default value.

## Informational

### [I-1] The `TSwapPool::PoolFactory__PoolDoesNotExist` error is not used and should be removed

```
- error PoolFactory__PoolDoesNotExist(string poolName);
```

### [I-2] Lacking zero address checks

```
constructor(address wethToken) {
+    if(wethToken == address(0)) {
+        revert("TSwapPool::constructor - wethToken is the zero address");
+    }
     i_wethToken = wethToken;
}
```

### [I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()` to get the token symbol

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

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

### [I-4] Magic numbers should not be used and should instead be replaced by constants

```
- uint256 inputAmountMinusFee = inputAmount * 997;
+ uint256 inputAmountMinusFee = inputAmount * FEE_DENOMINATOR;
uint256 numerator = inputAmountMinusFee * outputReserves;
- uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
+ uint256 denominator = (inputReserves * FEE_NUMERATOR) + inputAmountMinusFee;
return numerator / denominator;
```