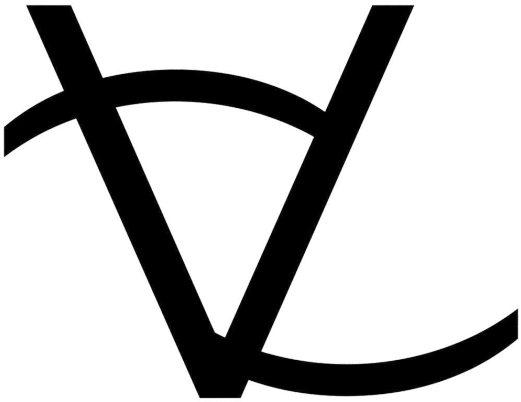

T-Swap Security Review

Initial Version

Valya Zaitseva

November 17, 2025

T-Swap Security Review

Valya Zaitseva

November 17, 2025

Prepared by: Valya Zaitseva

Table of Contents

- Medium
  - [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline
- 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__PoolDoesNotExist` error is not used
  - [I-2] Lacking zero address checks
  - [I-3] `PoolFactory::createPool` should use `.symbol()` instead of `name()`
  - [I-4] Use constants instead of 'magic numbers'

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.

Disclaimer

As a solo reviewer, I make every effort to identify as many vulnerabilities in the code within the given time period, but hold no responsibility for the findings provided in this document. A security audit 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

		Impact		
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

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda
- In Scope:

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

Issues found

Severity	Number of issues found
High	4
Medium	1

Severity	Number of issues found
Low	2
Info	4
Total	11

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 resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1000.

Impact: Protocol takes more fees than expected from users.

Proof of Concept: Actual input is ten times bigger than it should be, i.e. a user will need 10 times more input asset than it should be based on an invariant for input amount `delta x = (beta/(1-beta))* (1/gamma)* x` and 0.3% fee specified in docs for TSwap protocol.

```
1  function test_getInputAmountBasedOnOutput_inputAmountIsTenTimesBigger()
       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 inputReserves = poolToken.balanceOf(address(pool));
9          uint256 outputReserves = weth.balanceOf(address(pool));
10
11         uint256 outputAmount = 1e18;
12         poolToken.approve(address(pool), 10e18);
13         uint256 expectedInput = ((inputReserves * outputAmount) * 1000)
               / ((outputReserves - outputAmount) * 997);
14         uint256 actualInput = pool.getInputAmountBasedOnOutput(
              outputAmount, inputReserves, outputReserves);
15
```

```
16          assertEq(expectedInput * 10, actualInput);
17      }
```

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

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fever 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 transaction processes, the user could get a much worse swap.

Proof of Concept: 1. The price of 1 WETH right now is 10.131404313951956880 poolTokens. 2. User inputs a `swapExactOutput` looking for 1 WETH. 1. inputToken = USDC 2. outputToken = WETH 3. outputAmount = 1 3. The function does not offer a maxInputAmount 4. 10 swaps happened before this desired swap. 5. As the transaction is pending in the mempool, the market changes! And the price moves HUGE -> 1 WETH is not 10.131404313951956880 poolTokens. It is 31.281271380539048087 poolTokens. 6. The transaction completes, but the user sent the protocol 31.281271380539048087 poolTokens instead of the expected 10.131404313951956880

Recommended Mitigation: We should include a `maxInputAmount` so the user only has to spend up to a specific amount, and can anticipate how much they will spend.

```
1  + error TSwapPool__InputTooHight(uint256 inputAmount, uint256
       maxInputAmount);
```

```
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 TSwapPool__InputTooHight(inputAmount, maxInputAmount);
+    }
    _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 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: 1. Check how much output should be given based on the inputAmount of pool token. 2. Check actual result of `sellPoolTokens` function and compare it with result of `getInputAmountBasedOnOutput` function, where pool token amount is used mistakenly instead of output amount - they are equal. 3. Compare the actually received amount of weth token and the amount that should be - user receives more weth than it should be.

```
 function test_sellPoolTokens() 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);
        poolToken.approve(address(pool), 100e18);
        weth.approve(address(pool), 100e18);

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

        uint256 inputAmount = 1e18;

        uint256 shouldBeOutputAmountBasedOnInput =
            pool.getOutputAmountBasedOnInput(inputAmount, inputReserves
                , outputReserves);
        console.log("shouldBeOutputAmountBasedOnInput ",
            shouldBeOutputAmountBasedOnInput);

        uint256 actualResultBasedOnOutput = pool.
            getInputAmountBasedOnOutput(inputAmount, inputReserves,
            outputReserves);
        console.log("actualResultBasedOnOutput ",
            actualResultBasedOnOutput);

        uint256 actualResultBasedOnSellPoolTokens = pool.sellPoolTokens
            (inputAmount);
        console.log("actualResultBasedOnSellPoolTokens ",
            actualResultBasedOnSellPoolTokens);

        assertEq(actualResultBasedOnOutput,
            actualResultBasedOnSellPoolTokens);
        assertLt(shouldBeOutputAmountBasedOnInput,
            actualResultBasedOnSellPoolTokens);
    }
```

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 `minOutputAmount` to be passed to `swapExactInput`

```
-function sellPoolTokens(uint256 poolTokenAmount) external returns (
    uint256 wethAmount) {
+function sellPoolTokens(uint256 poolTokenAmount, uint256
    minOutputAmount) external returns (uint256 wethAmount) {
-       return swapExactOutput(i_poolToken, i_wethToken,
    poolTokenAmount, uint64(block.timestamp));
+       return swapExactInput(i_poolToken, poolTokenAmount,
    i_wethToken, minOutputAmount, uint64(block.timestamp));
}
```

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

[H-4] In `TSwapPool::_swap` the extra tokens given to users after every `SWAP_COUNT_MAX` 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 the 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 slowly over time the protocol funds will be drained.

The following block of code is responsible for the issue.

```
1
2  function _swap(IERC20 inputToken, uint256 inputAmount, IERC20
       outputToken, uint256 outputAmount) private {
3      if (_isUnknown(inputToken) || _isUnknown(outputToken) || inputToken
           == outputToken) {
4          revert TSwapPool__InvalidToken();
5      }
6
7  @>     swap_count++;
8  @>     if (swap_count >= SWAP_COUNT_MAX) {
9  @>         swap_count = 0;
10 @>         outputToken.safeTransfer(msg.sender, 1
       _000_000_000_000_000_000);
11     }
12     emit Swap(msg.sender, inputToken, inputAmount, outputToken,
           outputAmount);
```

Impact: A user could maliciously drain the protocol of funds by doing a lot of swaps and collection 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 until all the protocol funds are drained.

Proof Of Code

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

```
1  function test_invariantBreak() 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));
```

```
        vm.stopPrank();

        uint256 outputWeth = 1e17;

        // swap
        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)));
        // we are expecting our pool to change by expectedDelta..
            amounts
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);

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

        // actual deltaY after 'swapExactOutput'
        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);

        assertEq(actualDeltaY, expectedDeltaY);
    }
```

Recommended Mitigation: Remove the extra incentive. 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++;
-    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 to the documentation is “The deadline for the transaction to be completed by”. However, this parameter in never used.

As a consequence, operations 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:

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

Low

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

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran` 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:

```
- 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 0, giving incorrect information to the caller.

Proof of Concept:

```
 function test_swapExactInput_returnsZero() 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);
        poolToken.approve(address(pool), 10e18);
        // After we swap, there will be ~110 tokenA, and ~91 WETH
        // 100 * 100 = 10,000
        // 110 * ~91 = 10,000
        uint256 expected = 9e18;

        uint256 expectedOutput = pool.swapExactInput(poolToken, 10e18,
            weth, expected, uint64(block.timestamp));
        assertEq(expectedOutput, 0);
    }
```

Recommended Mitigation:

```
function swapExactInput(
    IERC20 inputToken, //input token to swap/sell, i.e.DAI
    uint256 inputAmount, // amount of input token to sell, i.e. DAI
    IERC20 outputToken, // WETH
    uint256 minOutputAmount, // e minimum output amount expected to
        receive
    uint64 deadline //when a tx expires
)
    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__PoolDoesNotExist` error is not used

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

[I-2] Lacking zero address 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());
```

[I-4] Use constants instead of 'magic numbers'

Description: There are numbers used in the code - it is better from gas usage point of view and more descriptive when constants are used

Impact: Using 'magic numbers' is prone of errors and not descriptive approach of code writing. For example in `getInputAmountBasedOnOutput` function magic numbers are used:

```
  function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        return ((inputReserves * outputAmount) * 10000) / ((
           outputReserves - outputAmount) * 997);
```

```
13      }
```

Recommended Mitigation: Use constants here:

```
1  uint256 public constant GAMMA = 997;
2  uint256 public constant PRECISION = 1000;
3  function getInputAmountBasedOnOutput(
4          uint256 outputAmount,
5          uint256 inputReserves,
6          uint256 outputReserves
7      )
8      public
9      pure
10     revertIfZero(outputAmount)
11     revertIfZero(outputReserves)
12     returns (uint256 inputAmount)
13 {
14 -     return ((inputReserves * outputAmount) * 10000) / ((outputReserves
       - outputAmount) * 997);
15 +     return ((inputReserves * outputAmount) * PRECISION) / ((
       outputReserves - outputAmount) * GAMMA);
16 }
17
18 function getOutputAmountBasedOnInput(
19     uint256 inputAmount,
20     uint256 inputReserves,
21     uint256 outputReserves
22 )
23     public
24     pure
25     revertIfZero(inputAmount)
26     revertIfZero(outputReserves)
27     returns (uint256 outputAmount)
28 {
29 -     uint256 inputAmountMinusFee = inputAmount * 997;
30 +     uint256 inputAmountMinusFee = inputAmount * GAMMA;
31     uint256 numerator = inputAmountMinusFee * outputReserves;
32 -     uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee
       ;
33 +     uint256 denominator = (inputReserves * PRECISION) +
       inputAmountMinusFee;
34     return numerator / denominator;
35 }
```

If constants were used, there would be no place for the error with the precision, when wrong precision is used in `getInputAmountBasedOnOutput`.