TSwap Protocol Audit Report

Version 1.0

DeniyalDaniDan

June 19, 2025

TSwap Protocol Audit Report

deniyaldanidan

June 19, 2025

Prepared by: DeniyalDaniDan

Lead Auditors:

- DeniyalDaniDan

Table of Contents

Protocol Summary

TSwap 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 **DeniyalDaniDan** 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

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

```
e643a8d4c2c802490976b538dd009b351b1c8dda
```

Click here to view the Source code

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

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
Low	2
Info	5
Gas	2
Total	14

Findings

High

[H-1] Incorrect Fee Calculation in `TSwapPool::getInputAmountBasedOnOutput()` causes protocol to take 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 tokens of output tokens. However, the function currently miscalculates the resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact:

Protocol takes more fees than expected from users.

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 potentialy receive way fewer 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 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 the 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:

paste the following function in the `TSwapPool.t.sol` file's `TSwapPoolTest` test contract.

1. Supply the pool initially with 200PoolTokens / 50 WETH
2. Make user sell 10 of his PoolTokens using `TSwapPool::sellPoolTokens()` function.
3. If you notice the user's final balance, he transferred more than 10 PoolTokens to the pool for exactly 10 WETH from the pool.

```
function test_proveSellPoolTokensBug() public {
        vm.startPrank(liquidityProvider);
        // we're gonna supply 200PT/50WETH
        uint256 initialPoolTokenToDeposit = 200 ether;
        uint256 initialWethToDeposit = 50 ether;
        poolToken.approve(address(pool), initialPoolTokenToDeposit);
        weth.approve(address(pool), initialWethToDeposit);

        pool.deposit(
            initialWethToDeposit,
            1 ether,
            initialPoolTokenToDeposit,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        // Check if deposit success?
        uint256 initialPoolTokenInPool = poolToken.balanceOf(address(
            pool));
        uint256 initialWethInPool = weth.balanceOf(address(pool));

        console.log("Initial Pool's PoolToken & WETH: ");
        console.log(initialPoolTokenInPool, initialWethInPool); // 100
            PT & 10 WETH

        assertEq(initialPoolTokenToDeposit, initialPoolTokenInPool);
        assertEq(initialWethToDeposit, initialWethInPool);

        address myUser = makeAddr("my-user");

        poolToken.mint(myUser, 1000 ether);
        weth.mint(myUser, 1000 ether);

        // check how much poolTokens & weth the myUser initially have??

        uint256 initialUserPoolToken = poolToken.balanceOf(myUser);
        uint256 initialUserWeth = weth.balanceOf(myUser);

        console.log("Initial User's PoolToken & WETH: ");
        console.log(initialUserPoolToken, initialUserWeth); // 1000 PT
            & 1000 WETH

        // Now myUser is gonna sell 10 of his poolToken for his Weth.
        uint256 poolTokensToSell = 10 ether;
        vm.startPrank(myUser);
        poolToken.approve(address(pool), type(uint256).max);
        pool.sellPoolTokens(poolTokensToSell);
        vm.stopPrank();

        uint256 finalUserPoolToken = poolToken.balanceOf(myUser);
```

```
48          uint256 finalUserWeth = weth.balanceOf(myUser);
49
50          uint256 finalPoolTokenInPool = poolToken.balanceOf(address(pool
                ));
51          uint256 finalWethInPool = weth.balanceOf(address(pool));
52
53          console.log("Final User's PoolToken & WETH: ");
54          console.log(finalUserPoolToken, finalUserWeth);
55          console.log("Final Pool's PoolToken & WETH: ");
56          console.log(finalPoolTokenInPool, finalWethInPool);
57
58          // because of the bug in the sellPoolTokens function he's gonna
                 gain 10 WETH for more than 10 POOLTOKEN
59          // This means sellPoolTokens() function gives 10 weth for X
                amount of PoolTokens instead of giving X amount of WETH for
                10 PoolTokens
60
61          assert((initialUserPoolToken - finalUserPoolToken) > 10 ether);
                 // he lost more than 10 PoolTokens
62          assert((finalUserWeth - initialUserWeth) == 10 ether); // he
                gained exactly 10 WETH
63
64          assert((finalPoolTokenInPool - initialPoolTokenInPool) > 10
                ether); // pool gained more than 10 PT from myUser
65          assert((initialWethInPool - finalWethInPool) == 10 ether); //
                pool lose exactly 10 WETH
66      }
```

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`)

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

[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 until all the protocol funds are drained

Proof Of Code:

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

```
function test_invariantBreak() public {
        vm.startPrank(liquidityProvider);
        // we're gonna supply 200PT/50WETH
        uint256 initialPoolTokenToDeposit = 200 ether;
        uint256 initialWethToDeposit = 50 ether;
        poolToken.approve(address(pool), initialPoolTokenToDeposit);
        weth.approve(address(pool), initialWethToDeposit);

        pool.deposit(
            initialWethToDeposit,
            1 ether,
            initialPoolTokenToDeposit,
            uint64(block.timestamp)
        );
```

```
15          vm.stopPrank();
16
17          address myUser = makeAddr("my-user");
18
19          poolToken.mint(myUser, 1000 ether);
20          weth.mint(myUser, 10 ether);
21
22          uint256 outputAmount = 1e16;
23
24          // 1st Swap
25          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 1);
26          // 2nd Swap
27          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 2);
28          // 3rd Swap
29          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 3);
30          // 4th Swap
31          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 4);
32          // 5th Swap
33          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 5);
34          // 6th Swap
35          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 6);
36          // 7th Swap
37          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 7);
38          // 8th Swap
39          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 8);
40          // 9th Swap
41          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 9);
42          // 10th Swap
43          _simulateSwapExactOutputAndAssertInvariant(myUser, outputAmount
                , 10);
44      }
45
46      function _simulateSwapExactOutputAndAssertInvariant(
47          address _userAddr,
48          uint256 _neededOutputAmount,
49          uint256 _swapCount
50      ) private {
51          console.log("Swap Count: #", _swapCount);
52          uint256 wethBalanceOfPoolBeforeSwap = weth.balanceOf(address(
                pool));
53          // start the swap
54          vm.startPrank(_userAddr);
```

```
55          poolToken.approve(address(pool), type(uint256).max);
56          pool.swapExactOutput(
57              poolToken,
58              weth,
59              _neededOutputAmount,
60              uint64(block.timestamp)
61          );
62          vm.stopPrank();
63
64          uint256 wethBalanceOfPoolAfterSwap = weth.balanceOf(address(
                pool));
65
66          // calculate deltaY's
67          int256 expectedDeltaY = int256(-1) * int256(_neededOutputAmount
                );
68          int256 actualDeltaY = int256(wethBalanceOfPoolAfterSwap) -
69              int256(wethBalanceOfPoolBeforeSwap);
70          // assert them
71          assertEq(expectedDeltaY, actualDeltaY);
72      }
```

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.

```
1 -         swap_count++;
2 -         // Fee-on-transfer
3 -         if (swap_count >= SWAP_COUNT_MAX) {
4 -             swap_count = 0;
5 -             outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
6 -         }
```

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 "the deadline for the transaction to be completed by". However, this parameter is never used. As a consquence, operations that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavourable.

Impact:

Transactions could be sent when market's rates are unfavorable to the deposit, even when adding the deadline parameter.

Proof of Concept:

The `deadline` parameter is unused.

Recommended Mitigation:

make following changes to the `TSwapPool::deposit()` function

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

Low

[L-1] Incorrect implementation of `TSwapPool::LiquidityAdded` event, causing event to emit incorrect implementation

Description:

When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTransfer()` function, it is incorrectly implemented. The `wethToDeposit` should come second and `poolTokensToDeposit` should be comes third.

Impact:

Event emission is incorrect, which can leads to off-chain functions to malfunction.

Recommended Mitigation:

```
    function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
    ) private {
        _mint(msg.sender, liquidityTokensToMint);
-       emit LiquidityAdded(msg.sender, wethToDeposit,
    poolTokensToDeposit);
```

```
+       emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
        /// remaining code....
```

[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.

Recommended Mitigation:

```
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

-       uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount, inputReserves, outputReserves);
+       output = getOutputAmountBasedOnInput(inputAmount, inputReserves, outputReserves);

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

-       _swap(inputToken, inputAmount, outputToken, outputAmount);
+       _swap(inputToken, inputAmount, outputToken, output);
}
}
```

Infos

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

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

[I-2] Lacking Zero Address checks in `constructor()` of PoolFactory Contract

```
    constructor(address wethToken) {
+       if (wethToken == address(0)){
+           revert("zero address is supplied");
+       }
        i_wethToken = wethToken;
    }
```

[I-3] Incorrectly used .name() instead of .symbol() in `PoolFactory::createPool()` function

```
    function createPool(address tokenAddress) external returns (address
        ) {
        if (s_pools[tokenAddress] != address(0)) {
            revert PoolFactory__PoolAlreadyExists(tokenAddress);
        }
        string memory liquidityTokenName = string.concat(
            "T-Swap ",
            IERC20(tokenAddress).name()
        );

        string memory liquidityTokenSymbol = string.concat(
            "ts",
-           IERC20(tokenAddress).name()
+           IERC20(tokenAddress).symbol()
        );
    // remaining code...
```

[I-4] Missing address checks in `constructor()` of TSwapPool contract

```
    constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
+       if (poolToken == address(0)){
+           revert("Zero address is supplied for poolToken");
+       }
        i_wethToken = IERC20(wethToken);
+       if (wethToken == address(0)){
+           revert("Zero address is supplied for wethToken")
```

```
+        }
         i_poolToken = IERC20(poolToken);
     }
```

[I-5] usage of magic numbers in `TSwapPool::getOutputAmountBasedOnInput()`

Description:

Magic numbers 997 and 1000 are used in `getOutputAmountBasedOnInput()` function. Usage of magic numbers in code is discouraged to avoid poor code readability and other potential issues

Recommended Mitigation:

```
+ uint256 private constant FEE_FACTOR = 997;
+ uint256 private constant FEE_SCALE = 1000;

/// remaining code ....

function getOutputAmountBasedOnInput(
        uint256 inputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(inputAmount)
        revertIfZero(outputReserves)
        returns (uint256 outputAmount)
    {
-       uint256 inputAmountMinusFee = inputAmount * 997;
+       uint256 inputAmountMinusFee = inputAmount * FEE_FACTOR;
        uint256 numerator = inputAmountMinusFee * outputReserves;
-       uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
+       uint256 denominator = (inputReserves * FEE_SCALE) + inputAmountMinusFee;
        return numerator / denominator;
    }
```

Gas

[G-1] unused variable in `TSwapPool::deposit()` function

Description:

`poolTokenReserves` variable is unused so remove that line from the `deposit()` function.

```
    if (totalLiquidityTokenSupply() > 0) {
        uint256 wethReserves = i_wethToken.balanceOf(address(this));
-        uint256 poolTokenReserves = i_poolToken.balanceOf(address(this
    ));
    /// Remaining code....
```

[G-2] `TSwapPool::swapExactInput()` function should be `external` not `public`

```
    function swapExactInput( IERC20 inputToken, uint256 inputAmount,
       IERC20 outputToken, uint256 minOutputAmount, uint64 deadline )
-       public
+       external
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
    /// Remaining code...
```