Tswap Protocol Audit Report

Version 1.0

November 4, 2025

Tswap Protocol Audit Report

SIDDU03

NOV 3, 2025

Prepared by: [SIDDU03] Lead Auditors: - SIDDU MULKALLA

Table of Contents

    * [H-5] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`
  - Medium
    * [M-1] Missing deadline check when adding liquidity
    * [M-2]Lack of slippage protection in `swapExactOutput` function
  - 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::PoolFactory__PoolDoesNotExist` is not used and should be removed
    * [I-2] Lacking zero address checks
    * [I-3] `PoolFacotry::createPool` should use `.symbol()` instead of `.name()`
    * [I-4] Event is missing `indexed` fields # Protocol Summary

TSwap

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

TSwap Pools

The protocol starts as simply a `PoolFactory` contract. This contract is used to create new "pools" of tokens. It helps make sure every pool token uses the correct logic. But all the magic is in each `TSwapPool` contract.

You can think of each `TSwapPool` contract as it's own exchange between exactly 2 assets. Any ERC20 and the WETH token. These pools allow users to permissionlessly swap between an ERC20 that has a pool and WETH. Once enough pools are created, users can easily "hop" between supported ERC20s.

For example: 1. User A has 10 USDC 2. They want to use it to buy DAI 3. They `swap` their 10 USDC -> WETH in the USDC/WETH pool 4. Then they `swap` their WETH -> DAI in the DAI/WETH pool

Every pool is a pair of `TOKEN X` & `WETH`.

There are 2 functions users can call to swap tokens in the pool. - `swapExactInput` - `swapExactOutput`

We will talk about what those do in a little.

Liquidity Providers

In order for the system to work, users have to provide liquidity, aka, “add tokens into the pool”.

Why would I want to add tokens to the pool?

The TSwap protocol accrues fees from users who make swaps. Every swap has a `0.3` fee, represented in `getInputAmountBasedOnOutput` and `getOutputAmountBasedOnInput`. Each applies a `997` out of `1000` multiplier. That fee stays in the protocol.

When you deposit tokens into the protocol, you are rewarded with an LP token. You’ll notice `TSwapPool` inherits the `ERC20` contract. This is because the `TSwapPool` gives out an ERC20 when Liquidity Providers (LP)s deposit tokens. This represents their share of the pool, how much they put in. When users swap funds, 0.03% of the swap stays in the pool, netting LPs a small profit.

LP Example

1. LP A adds 1,000 WETH & 1,000 USDC to the USDC/WETH pool
    1. They gain 1,000 LP tokens
2. LP B adds 500 WETH & 500 USDC to the USDC/WETH pool
    1. They gain 500 LP tokens
3. There are now 1,500 WETH & 1,500 USDC in the pool
4. User A swaps 100 USDC -> 100 WETH.
    1. The pool takes 0.3%, aka 0.3 USDC.
    2. The pool balance is now 1,400.3 WETH & 1,600 USDC
    3. aka: They send the pool 100 USDC, and the pool sends them 99.7 WETH

Note, in practice, the pool would have slightly different values than 1,400.3 WETH & 1,600 USDC due to the math below.

Core Invariant

Our system works because the ratio of Token A & WETH will always stay the same. Well, for the most part. Since we add fees, our invariant technially increases.

`x * y = k` - x = Token Balance X - y = Token Balance Y - k = The constant ratio between X & Y

Our protocol should always follow this invariant in order to keep swapping correctly!

Make a swap

After a pool has liquidity, there are 2 functions users can call to swap tokens in the pool. - `swapExactInput` - `swapExactOutput`

A user can either choose exactly how much to input (ie: I want to use 10 USDC to get however much WETH the market says it is), or they can choose exactly how much they want to get out (ie: I want to get 10 WETH from however much USDC the market says it is.)

This codebase is based loosely on Uniswap v1

Disclaimer

The Siddu03 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
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

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda

Scope

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

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.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	5
Gas	0
Total	13

Findings

High

[H-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 is never used. As a consequence, operationrs 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, // LP tokens -> if empty, we can pick 100% (100% == 17 tokens)
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
```

[H-2] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocll 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 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) * 10_000) / ((
    outputReserves - outputAmount) * 997);
+        return ((inputReserves * outputAmount) * 1_000) / ((
    outputReserves - outputAmount) * 997);
    }
```

[H-3] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially 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 transaciton 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

Proof Of Code

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

```

    function testFeesCollectedIsGreaterThanWhatMentioned() public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 200e18);
    poolToken.approve(address(pool), 200e18);
    pool.deposit(200e18, 100e18, 200e18, uint64(block.timestamp));
    vm.stopPrank();

    vm.startPrank(user);

    uint256 starting_balance_of_user = weth.balanceOf(address(user));
```

```
12
13      console.log(poolToken.balanceOf(address(pool)));
14      console.log(weth.balanceOf(address(pool)));
15
16      uint256 expectedInputAmount = ((weth.balanceOf(address(pool)) * (10
            e18))
17          * 10000 / ((poolToken.balanceOf(address(pool)) - (10e18)) *
                997));
18
19      uint256 testing = pool.getInputAmountBasedOnOutput(
20          10e18,
21          weth.balanceOf(address(pool)),
22          poolToken.balanceOf(address(pool))
23      );
24
25      console.log(testing);
26      console.log(starting_balance_of_user);
27
28      console.log(expectedInputAmount);
29      uint256 expectedBalanceOfUserAfterTransaction =
30          starting_balance_of_user - expectedInputAmount;
31
32      console.log(expectedBalanceOfUserAfterTransaction);
33
34      weth.approve(address(pool), 200e18);
35      pool.swapExactOutput(weth, poolToken, 10e18, uint64(block.timestamp
            ));
36      vm.stopPrank();
37
38      console.log(weth.balanceOf(address(user)));
39
40      assert(
41          weth.balanceOf(address(user)) <
42          expectedBalanceOfUserAfterTransaction
43      );
44  }
```

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.

```
1       function swapExactOutput(
2           IERC20 inputToken,
3   +       uint256 maxInputAmount,
4   .
5   .
6   .
7           inputAmount = getInputAmountBasedOnOutput(outputAmount,
                inputReserves, outputReserves);
8   +       if(inputAmount > maxInputAmount){
9   +           revert();
10  +       }
```

```
11          _swap(inputToken, inputAmount, outputToken, outputAmount);
```

[H-4] `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 miscalculaes 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 protcol 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 (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      }
```

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

[H-5] 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 untill all the protocol funds are drained

Proof Of Code

Place the following into TSwapPool.t.sol.

```

    function testInvariantBroken() 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));
```

```
20          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
21          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
22          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
23
24          int256 startingY = int256(weth.balanceOf(address(pool)));
25          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
26
27          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
28          vm.stopPrank();
29
30          uint256 endingY = weth.balanceOf(address(pool));
31          int256 actualDeltaY = int256(endingY) - int256(startingY);
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

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] Missing deadline check when adding liquidity

The `deposit` function accepts a `deadline` parameter, which according to documentation is “The deadline for the transaction to be completed by”. However, this parameter is 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 for the caller.

Consider making the following change to the `deposit` function:

```
1      function deposit(
2          uint256 wethToDeposit,
3          uint256 minimumLiquidityTokensToMint,
4          uint256 maximumPoolTokensToDeposit,
5          uint64 deadline
```

```
    )
        external
        revertIfZero(wethToDeposit)
+       revertIfDeadlinePassed(deadline)
        returns (uint256 liquidityTokensToMint)
```

[M-2] Lack of slippage protection in `swapExactOutput` function

The `swapExactOutput` function does not include any sort of slippage protection to protect user funds that swap tokens in the pool. Similar to what is done in the `swapExactInput` function, it should include a parameter (e.g., `maxInputAmount`) that allows callers to specify the maximum amount of tokens they're willing to pay in their trades.

```
function swapExactOutput(
    IERC20 inputToken,
+   uint256 maxInputAmount
    IERC20 outputToken,
    uint256 outputAmount,
    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__OutputTooHigh(inputAmount, maxInputAmount);
+   }

    _swap(
        inputToken,
        inputAmount,
        outputToken,
        outputAmount
    );
}
```

Lows

[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 `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 `ouput` it is never assigned a value, nor uses an explict 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);

    }
```

```
18      }
```

Informationals

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

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

[I-2] lacking Zero address check

```
  constructor(address wethToken) {
+       if(wethToken == address(0)){
+        revert();
+       }
        i_wethToken = wethToken;
    }
```

[I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`

```
-   string memory liquidityTokenName = string.concat("T-Swap ", IERC20(tokenAddress).name());
-   string memory liquidityTokenSymbol = string.concat("ts", IERC20(tokenAddress).name());
+   string memory liquidityTokenName = string.concat("T-Swap ", IERC20(tokenAddress).symbol());
+    string memory liquidityTokenSymbol = string.concat("ts", IERC20(tokenAddress).symbol());
```

[I-4] `TswapPool::Swap()` if there are more than 4 parameters in even then 3 should be indexed.

```
event Swap(
        address indexed swapper,
-       IERC20 tokenIn,
+       IERC20 indexed tokenIn,
-       uint256 amountTokenIn,
+       uint256 indexed amountTokenIn,
        IERC20 tokenOut,
        uint256 amountTokenOut
    );
```

[I-5] Provide better error for `TSwapPool::getInputAmountBasedOnOutput`

When `outputReserves` & `outputAmount` are the same, `TSwapPool::getInputAmountBasedOnOutput` will revert with arithmetic divide by zero. This is not very helpful for users. Consider adding a custom error message.

```
+       error TSwapPool__CannotRemoveEntireBalance();
.
.
.
+       if (outputReserves, outputAmount) {revert
    TSwapPool__CannotRemoveEntireBalance();}
        return (inputReserves * outputAmount) / ((outputReserves -
            outputAmount));
```