TSwap Protocol Audit Report

Version 1.0

akkses.io

September 6, 2024

TSwap Protocol Audit Report

yudistira19

September 06, 2024

Prepared by: AKKSes Lead Auditors:

- Yudistira Adi Wahyudi R

Table of Contents

- High-Level Findings
  * [H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees
  * [H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens
  * [H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens
  * [H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`
- Medium-Level Findings
  * [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline
  * [M-2] Rebase, Fee-on-transfer, and ERC777 tokens break protocol invariant
- Low-Level Findings
  * [L-1] `TSwapPool::_addLiquidityMintAndTransfer` event has parameters out of order causing event to emit incorrect information.
  * [L-2] Default value returned by `TSwapPool::swapExactInput` result is incorrect return value given
- Informational
  * [I-1] Poor test coverage
  * [I-2] `PoolFactory::PoolFactory__PoolAlreadyExists` is not used and should be removed
  * [I-3] Lacking zero address check
  * [I-4] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`
  * [I-5] Event is missing `indexed` fields

Protocol Summary

T-Swap

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 technically increases.

`x * y = k`

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

```
y = Token Balance Y
x = Token Balance X
x * y = k
x * y = (x + Δx) * (y -Δ y)Δ
x = Change of token balance XΔ
y = Change of token balance Yβ
 = Δ(y / y)α
 = Δ(x / x)

Final invariant equation without fees:Δ
x = ββ(/(1-)) * xΔ
y = αα(/(1+)) * y

Invariant with feesρ
 = fee (between 0 & 1, aka a percentage)γ
 = (1 - p) (pronounced gamma)Δ
x = ββ(/(1-)) * γ(1/) * xΔ
y = αγαγ(/1+) * 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 AKkses 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
	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

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda

Scope

- 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

I Love this course from updraft cyfrin, teach me many thing. Thanks to Patrick Collins for the amazing video. This is my second auditing, still have to much learn and learning again.

Issues found

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

Findings

High-Level Findings

[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 calculated 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.

Proof of Concept:

1. Liquidity Provider Setup:
    - The liquidity provider starts by depositing 100 WETH and 100 pool tokens into the pool, setting up an initial 1:1 liquidity ratio.
2. User Mint and Swap:
    - A user is minted 11 pool tokens and attempts to swap for 1 WETH from the pool.
    - Due to the flaw, the user spends almost all of their pool tokens (far more than the expected 1 pool token) to get 1 WETH.
3. Rug Pull by Liquidity Provider:
    - After the user's swap, the liquidity provider can withdraw all funds from the pool, including the user's pool token contribution.
    - The pool's WETH and pool token balances are reduced to zero, and the liquidity provider ends up with all the remaining assets, leaving the user with far less value than they originally had.

Proof Of Code

```
    function testFlawedSwapExactOutput() public {
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // User has 11 pool tokens
        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 11e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);
        vm.startPrank(someUser);

        // Users buys 1 WETH from the pool, paying with pool tokens
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, 1 ether, uint64(block.
            timestamp));

        // Initial Liquidity was 1:1, so user should have paid ~1 pool
            token
        // However, it spent much more than that. The user started with
             11 tokens, and now only has less than 1

        assertLt(poolToken.balanceOf(someUser), 1 ether);
        vm.stopPrank();

        // The liquidity provider can rug all fund from the pool now,
        // including those deposited by the user
        vm.startPrank(liquidityProvider);
        pool.withdraw(
            pool.balanceOf(liquidityProvider),
            1, // minWethToWithdraw
            1, // minPoolTokenToWithdraw
            uint64(block.timestamp)
        );
        assertEq(weth.balanceOf(address(pool)), 0);
        assertEq(poolToken.balanceOf(address(pool)), 0);
        // vm.stopPrank();
    }
```

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-2] 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 condition change before the transaction, the users 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 WETH
    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

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IERC20 {
    function transferFrom(address sender, address recipient, uint256
        amount) external returns (bool);
    function transfer(address recipient, uint256 amount) external
        returns (bool);
    function balanceOf(address account) external view returns (uint256)
        ;
    function approve(address spender, uint256 amount) external returns
        (bool);
}

contract VulnerableSwap {

    IERC20 public usdc;  // USDC token
    IERC20 public weth;  // WETH token
    uint256 public wethPriceInUsdc; // WETH price in USDC (1 WETH = x
        USDC)

    constructor(address _usdc, address _weth) {
        usdc = IERC20(_usdc);
        weth = IERC20(_weth);
        wethPriceInUsdc = 1000 * 10 ** 18; // Initial price: 1 WETH =
            1,000 USDC
    }

    // Simulate market change
    function changeWethPrice(uint256 newPriceInUsdc) external {
        wethPriceInUsdc = newPriceInUsdc;
    }

    // Vulnerable swapExactOutput function
    function swapExactOutput(uint256 outputAmount) external {
        // No maxInput to protect against slippage
        // Calculate how much USDC the user must send
        uint256 requiredUsdc = outputAmount * wethPriceInUsdc / 1 ether
            ;

        // Transfer USDC from user to contract
        usdc.transferFrom(msg.sender, address(this), requiredUsdc);

        // Transfer WETH to the user
        weth.transfer(msg.sender, outputAmount);
    }
}
```

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 tokens, the function will not be able to calculate the correct amount of output tokens to receive.

Impact: Users will swap the wrong amount of tokens, which is a severe disruption to the protocol functionality.

Proof of Concept:

Recommended Mitigation:

Consider changing the implementation to use the `swapExactInput` function instead of the `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (ie `minWethToReceive` to be passed to the `swapExactInput` function).

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

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

[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 follow a strict invariant of `x * y = k`, and the invariant is broken when the `swapCount` parameter is changed. 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 from the pool.

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

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

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

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

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.

```
-        swap_count++;
-        if (swap_count >= SWAP_COUNT_MAX) {
-            swap_count = 0;
-            outputToken.safeTransfer(msg.sender, 1
    _000_000_000_000_000_000);
-        }=
```

Medium-Level Findings

[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 is never used. As a consequence, operations that add liquidity to the pool might be executed at unexpected times, ini 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)
    {}
```

[M-2] Rebase, Fee-on-transfer, and ERC777 tokens break protocol invariant

Description: The core invariant of the protocol is:

`x * y = k`. In practice though, the protocol takes fees and actually increases k. So we need to make sure `x * y = k` before fees are applied.

Impact:

Proof of Concept:

Low-Level Findings

[L-1] `TSwapPool::_addLiquidityMintAndTransfer` event has parameters out of order causing event to emit incorrect information.

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran`
function, it logs values in an incorrect order. The `poolTokensToDeposit` parameter should go in the third position, where the `wethToDeposit` value should go in the second parameter.

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` result is 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: // have to practice with this one

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

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

Informational

[I-1] Poor test coverage

```
Running tests...
| File                | % Lines          | % Statements     | % Branches
      | % Funcs         |
| ------------------- | ---------------- | ---------------- |
  -------------- | -------------- |
| src/PoolFactory.sol | 100.00% (11/11) | 100.00% (16/16) | 100.00%
   (2/2) | 100.00% (3/3) |
| src/TSwapPool.sol   | 54.84% (34/62)  | 59.14% (55/93)  | 33.33%
   (6/18) | 37.50% (6/16) |
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.

[I-2] `PoolFactory::PoolFactory__PoolAlreadyExists` is not used and should be removed

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

[I-3] Lacking zero address check

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

[I-4] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`

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

[I-5] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

4 Found Instances

- Found in src/PoolFactory.sol Line: 35

  ```
      event PoolCreated(address tokenAddress, address poolAddress);
  ```

- Found in src/TSwapPool.sol Line: 52

  ```
      event LiquidityAdded()
  ```

- Found in src/TSwapPool.sol Line: 57

  ```
      event LiquidityRemoved()
  ```

- Found in src/TSwapPool.sol Line: 62

  ```
      event Swap()
  ```