PHOENIX
SMART SECURITY
BY TANU GUPTA

TSwap Protocol Audit Report

Version 1.0

Tanu Gupta

July 22, 2025

TSwap Protocol Audit Report

Tanu Gupta

July 22, 2025

Prepared by: Tanu Gupta

Lead Security Researcher:

- Tanu Gupta

Table of Contents

    * [H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protcol to take too many tokens from the user, resulting in a loss of funds
    * [H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to spend way more tokens
    * [H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive incorrect amount of tokens
    * [H-4] In `TSWapPool::_swap` the extra tokens given to users after every `swapCount` reaches 10 or more, breaks the invariant of `x * y = k`
  - Medium
    * [M-1] `TSwapPool::deposit` is missing a `deadline` parameter check causing transactions to be processed after the deadline
    * [M-2] Rebase, fee-on-tranfer and ERC-777 break protocol invariant `x * y = k`
  - Low
    * [L-1] The arguments set for this event `TSwapPool::LiquidityAdded` are not set in correct order
    * [L-2] Default value retuned by `TSwapPool::swapExactInput` results in incorrect value being returned
  - Informational
    * [I-1] Unused custom error `PoolFactory::PoolFactory__PoolDoesNotExist` in `PoolFactory` contract
    * [I-2] `wethToken` parameter in the constructor of the `PoolFactory` contract is not checked for zero address
    * [I-3] `liquidityTokenSymbol` variable of `PoolFactory::createPool` function should use `symbol()` instead of `name()`
    * [I-4] `deadline` parameter in `TSwapPool::deposit` function is not used
    * [I-5]: Event is missing `indexed` fields
    * [I-6] `TSwapPool::TSwapPool__WethDepositAmountTooLow` in this event, `TSwapPool::MINIMUM_WETH_LIQUIDITY` is not required to be emitted as this a constant value
    * [I-7] `TSwapPool::deposit` function is not following CEI for the first-time liquidity deposit
    * [I-8] Usage of magic numbers such as 997 and 1000 is not recommended
    * [I-9] Natspec is not defined for the function `TSwapPool::swapExactInput`
    * [I-10] `TSwapPool::swapExactOutput` natspec is missing the `@param` tag for `deadline`
  - Gas
    * [G-1] `TSwapPool::swapExactInput` and `TSwapPool::totalLiquidityTokenSupply`

functions should be marked as `external` instead of **`public`**

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.

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.

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

There are 2 functions users can call to swap tokens in the pool.

- `swapExactInput`
- `swapExactOutput`

Core Invariant - `x * y = k`

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

This codebase is based loosely on Uniswap v1

Disclaimer

The 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

The findings described in this document correspond the following github repository t-swap

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

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

Found the bugs using a tool called foundry including invariant fuzzing.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	10
Gas	1
Total	19

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protcol to take too many tokens from the user, resulting in a loss of funds

Description: The `getInputAmountBasedOnOutput` function in the `TSwapPool` contract calculates the input amount based on the output amount and reserves, but it uses a fee calculation that is not aligned with the expected behavior. When calculating the fee, it scales the amount by 10_000 instaed of 1000.

Impact: The fee is applied incorrectly, leading to the protocol taking more tokens from the user than intended. **Proof of code**

```
function test_UserIsChargedMoreThanIntended() external {
        uint256 initialWethAmount = 100e18;
        uint256 initialPoolTokenAmount = 100e18;
        //initial liquidity added to the pool
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), type(uint256).max);
        poolToken.approve(address(pool), type(uint256).max);
        pool.deposit(initialWethAmount, 0, initialPoolTokenAmount, uint64(block.timestamp));
        vm.stopPrank();

        //user wants to sell WETH tokens
        uint256 wethToBuy = 1e18;
        //poolTokens expected to receive
        //since the pool ratio is 1:1, the user is expected to spend ~ 1 pool token
        uint256 expectedPoolTokensToAdd = pool.getInputAmountBasedOnOutput(wethToBuy, poolToken.balanceOf(address(pool)), weth.balanceOf(address(pool)));
        console.log("Expected PoolTokens to add:", expectedPoolTokensToAdd);
        uint256 inputReserve = poolToken.balanceOf(address(pool));
        uint256 outputReserve = weth.balanceOf(address(pool));
        uint256 outputAmount = wethToBuy;

        uint256 actualPoolTokensToAdd = (1000 * (inputReserve * outputAmount)) / (997 * (outputReserve - outputAmount));
        console.log("Actual PoolTokens to add:", actualPoolTokensToAdd);

        assertLt(actualPoolTokensToAdd, expectedPoolTokensToAdd, "User is charged more than intended");

        //initiate the swap
        address someUser = makeAddr("someUser");
        poolToken.mint(someUser, 11 ether); //minting more than expected to ensure the user has enough balance
        vm.startPrank(someUser);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, wethToBuy, uint64(block.timestamp));
        vm.stopPrank();

        uint256 poolTokensAfterSwap = poolToken.balanceOf(someUser);
        //after swap the user is left with less than 1 pool token (i.e spent nearly 10 times of the intended ~10 pool tokens), when they should have spent nearly 1 pool token for the purchase of 1 WETH
        assertLt(poolTokensAfterSwap, 1e18, "User should have spent less than 1 pool token");

    }
```

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 spend way more tokens

Description: The `swapExactOutput` function does not include slippage protection, which means that users can end up spending significantly more tokens than they expect. The function calculates the input amount based on the output amount and reserves, but it does not check if the input amount exceeds a certain threshold.

This function is similar to `TSwapPool::swapExactInput` where the function specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount` to protect against slippage.

Impact: If the market conditions change between the time the user initiates the swap and the time it is executed, the user may end up spending much more than they intended, leading to potential loss of funds.

Proof of Concept:

1. The price of 1 WETH is 1 USDC.
2. The user wants to buy 1 WETH using pool tokens.
3. User inputs the `swapExactOutput` looking to buy 1 WETH, but does not specify a maximum amount of pool tokens they are willing to spend.
    - inputToken = poolToken
    - outputToken = weth
    - outputAmount = 1WETH
    - deadline = whatever
4. As the transaction is waiting in the mempool, the market changes.

5. Hence, the user ends up spending nearly 10 pool tokens for 1 WETH as the function does not have slippage protection.

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

        uint256 wethToBuy = 1e18;
        //user doesn't want to spend more than 2e18 pool tokens for 1
            e18 WETH
        //since the pool ratio is 1:1, the user is expected to spend ~
            1 pool token
        //initiate the swap
        address someUser = makeAddr("someUser");
        poolToken.mint(someUser, 11 ether); //minting more than
            expected to ensure the user has enough balance
        vm.startPrank(someUser);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, wethToBuy, uint64(block.
            timestamp));
        vm.stopPrank();

        uint256 poolTokensAfterSwap = poolToken.balanceOf(someUser);
        //after swap the user is left with less than 1 pool token due
            to price slippage, when they should have spent nearly 1 pool
             token for the purchase of 1 WETH
        assertLt(poolTokensAfterSwap, 1e18, "User should have spent
            less than 1 pool token");
    }
```

Recommended Mitigation: We should include a `maxInputAmount` parameter in the `swapExactOutput` function to allow users to set a limit on the maximum amount of input tokens they are willing to spend.

```
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__InputTooHigh(inputAmount, maxInputAmount)
    ;
+       }


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

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

Description: The `sellPoolTokens` function in the `TSwapPool` contract is designed to allow users to sell their pool tokens for a specified output token. Users indicate how many pool tokens they are willing to sell in the `poolTokenAmount` parameter. However, the function miscalculates the swapped amount.

This is due to the fact that `swapExactOutput` function is called whereas this should be `swapExactInput` called rather 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 the protocol's functionality.

Proof of Concept:

1. User has 10 WETH and 10 Pool tokens.
2. User wants to 10 Pool tokens to get ~9 WETH, totally to ~19 WETH in user's balance
3. Rather function tries to take these 10 Pool tokens as output tokens and expects user to have ~11 WETH for the transaction to go through.
4. Hence, reverts while performing the unintended operation.

```
function test_SellPoolTokens_Reverts() external {
        uint256 initialWethAmount = 100e18;
        uint256 initialPoolTokenAmount = 100e18;
        //initial liquidity added to the pool
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), type(uint256).max);
        poolToken.approve(address(pool), type(uint256).max);
        pool.deposit(initialWethAmount, 0, initialPoolTokenAmount,
            uint64(block.timestamp));
        vm.stopPrank();

        //user wants to sell pool tokens
        uint256 poolTokenToSell = 10e18;
        //weth expected to receive
        uint256 expectedWethToReceive = pool.
           getOutputAmountBasedOnInput(poolTokenToSell,
            poolToken.balanceOf(address(pool)), weth.balanceOf(address(
                pool)));
        console.log("Expected WETH to receive:", expectedWethToReceive)
           ;

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);

        vm.expectPartialRevert(IERC20Errors.ERC20InsufficientBalance.
           selector);

        pool.sellPoolTokens(poolTokenToSell);
        vm.stopPrank();
    }
```

Recommended Mitigation: Consider changing the implementation to use `swapExactInput` rather `swapExactOutput`. This would also require changing the `sellPoolTokens` function to accept a new parameter `minWethToReceive` to be passed to `swapExactInput`.

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

It might be wise to add a deadline to the function, as currently there is no deadline to curb MEV.

[H-4] In `TSwapPool::_swap` the extra tokens given to users after every swapCount reaches 10 or more, breaks the invariant of `x * y = k`

Description: The protocol follows a strict invariant of `x * y = k`, where:

- `x`: balance of pool token
- `y`: balance of weth
- `k`: constant product of two balances

This means 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 extra incentive in the `_swap` function.

Meaning that overtime the protocol funds will be drained.

```
//The following block of code is reposible 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 funds by doing a lot of swaps and collecting the extra incentive given out by the protocol.

More 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. The user continues to swap until all the protocol funds are drained.

Proof of Code

Place the following into TSwapPool.t

```
    function test_invariant_broke_x_product_y_not_constant() external {

        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;
        poolToken.mint(user, 100e18);

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        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(outputWeth) * int256(-1);

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

        vm.stopPrank();

        int256 endingY = int256(weth.balanceOf(address(pool)));

        int256 actualDeltaY = endingY - startingY;
        assertEq(expectedDeltaY, actualDeltaY);

    }
```

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 the tokens in the same way we do for 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 a `deadline` parameter check causing transactions to be processed after the deadline

Description: The `TSwapPool::deposit` function in the `TSwapPool` contract includes a `deadline` parameter which according to the documentation is **The deadline for the transaction to be completed by**, but there is no check to ensure that the transaction is processed before the deadline. This could allow users to execute **add liquidity** transactions at an unexpected time, leading to potential loss of funds when the deposit rate in unfavorable.

Impact: Transactions could be sent when market conditions are unfavourable to deposit even after adding a deadline parameter to the function without checking it could lead to unexpected behavior and potential loss of funds.

Proof of Concept: The `deadline` parameter is unused.

1. Producing Sandwich attack on deposit function causing user to receive less tokens than intended

Sandwich attack POC

```
function test_SandwichAttackOnDeposit() external {
        //initial liquidity added to the pool:
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        //user tries to make a deposit of 5e18 WETH
        uint256 wethAmountToDeposit = 5e18;
        uint256 poolTokenToDepositBeforeFrontRun = pool.
            getPoolTokensToDepositBasedOnWeth(wethAmountToDeposit);

        //attacker tries to front-run the deposit by making a swap
            trade
        uint256 outputWeth = 7e18;
        uint256 inputReserve_PoolToken = poolToken.balanceOf(address(
            pool));
        uint256 outputReserve_Weth = weth.balanceOf(address(pool));
        uint256 inputPoolToken =
             pool.getInputAmountBasedOnOutput(outputWeth,
                 inputReserve_PoolToken, outputReserve_Weth);
        // Record attacker's initial balances
        uint256 attackerWethBefore = weth.balanceOf(attacker);
        uint256 attackerPoolTokenBefore = poolToken.balanceOf(attacker)
            ;

        poolToken.mint(attacker, inputPoolToken + 1); //just to be sure

        vm.startPrank(attacker);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp));
        vm.stopPrank();

        //user makes a deposit
        uint256 poolTokenToDepositAfterFrontRun = pool.
            getPoolTokensToDepositBasedOnWeth(wethAmountToDeposit);
        assertGt(poolTokenToDepositAfterFrontRun,
            poolTokenToDepositBeforeFrontRun, "Front-run failed");

        vm.startPrank(user);
        require(poolToken.balanceOf(user) >=
            poolTokenToDepositAfterFrontRun, "Not enough pool tokens");
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.deposit(wethAmountToDeposit, 0,
            poolTokenToDepositAfterFrontRun, uint64(block.timestamp));
        vm.stopPrank();

        // BACKRUN: Attacker reverses the trade (WETH -> PoolToken)
        vm.startPrank(attacker);
        uint256 wethToSwapBack = outputWeth;
        weth.approve(address(pool), wethToSwapBack);
        pool.swapExactInput(weth, wethToSwapBack, poolToken, 0, uint64(
            block.timestamp));
        vm.stopPrank();

        // Calculate attacker's profit
        uint256 attackerWethAfter = weth.balanceOf(attacker);
        uint256 attackerPoolTokenAfter = poolToken.balanceOf(attacker);

        // Attacker should have more PoolToken than they started with
        int256 wethProfit = int256(attackerWethAfter) - int256(
            attackerWethBefore);
        int256 poolTokenProfit = int256(attackerPoolTokenAfter) -
            int256(attackerPoolTokenBefore);

        console.log("Attacker WETH profit:", wethProfit);
        console.log("Attacker PoolToken profit:", poolTokenProfit);
        console.log("User paid extra PoolToken:",
            poolTokenToDepositAfterFrontRun -
            poolTokenToDepositBeforeFrontRun);

        // The attacker should be profitable in at least one token
        assertTrue(poolTokenProfit > 0 || wethProfit > 0, "Attack
            should be profitable");
    }
```

Recommended Mitigation: Consider making the following changes to the function.

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

[M-2] Rebase, fee-on-tranfer and ERC-777 break protocol invariant `x * y = k`

Description: The TSwap AMM pool fails to account for various non-standard ERC20 token behaviors, causing systematic violations of the core AMM invariant `x * y = k`.

The pool's accounting system assumes standard ERC20 transfer behavior where the exact amount specified in `transferFrom()` is received by the pool.

However, multiple categories of **weird ERC20** tokens such as fee-on-tranfer, rebase, reentrant, pausable tokens, etc can cause discrepancies between expected and actual token balances, breaking the fundamental mathematical relationship that ensures proper price discovery, swap calculations, and liquidity management. For example -

Fee-on-Transfer Tokens:

- Pool expects: transferFrom(user, pool, 100e18) → pool receives 100e18
- Reality: Pool receives 100e18 - fee (e.g., 99e18)

Impact: `x * y = k` breaks, damaging the integrity of the AMM model.

Proof of Concept:

1. A user adds liquidity for a (weird ERC20 - fee on transfer) token.
2. Pool receives less than the intended liquidity for the given amount of WETH.
3. Hence, the invariants breaks.

Proof of Code

Place the following into TSwapPool.t

```
function test_invariant_broken_for_Weird_ERC20() external {
        TransferFeeToken newPoolToken = new TransferFeeToken(1e15, "
            Weird ERC20", "wERC20");

        weth = new ERC20Mock();
        pool = new TSwapPool(address(newPoolToken), address(weth), "
            LTokenA", "LA");

        weth.mint(liquidityProvider, 200e18);
        newPoolToken.mint(liquidityProvider, 200e18);

        weth.mint(user, 10e18);
        newPoolToken.mint(user, 10e18);

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        newPoolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 wethAmount = 5e18;
        int256 startingX = int256(newPoolToken.balanceOf(address(pool))
            );
        uint256 poolTokenAmount = pool.
            getPoolTokensToDepositBasedOnWeth(wethAmount);
        int256 expectedDeltaX = int256(poolTokenAmount);

        vm.startPrank(user);
        weth.approve(address(pool), wethAmount);
        newPoolToken.approve(address(pool), poolTokenAmount);
        pool.deposit(wethAmount, 0, poolTokenAmount, uint64(block.
            timestamp));
        vm.stopPrank();

        int256 endingX = int256(poolToken.balanceOf(address(pool)));
        int256 actualDeltaX = endingX - startingX;

        assertEq(expectedDeltaX, actualDeltaX, "Delta X mismatch");
    }
```

Recommended Mitigation: - Use balance delta checks before/after transferFrom and transfer to conform the integrity of AMM. - Only allow a known set of vetted tokens into pools, explicitly excluding known non-standard ones

Low

[L-1] The arguments set for this event `TSwapPool::LiquidityAdded` are not set in correct order

Description: The event `LiquidityAdded` is emitting the wrong arguments from function `TSwapPool::_addLiquidityMintAndTransfer`. The order of the arguments should be `wethDeposited` first, then `poolTokensDeposited`.

```
//event definition
event LiquidityAdded(address indexed liquidityProvider, uint256
    wethDeposited, uint256 poolTokensDeposited);
```

```
//event emission
emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
```

Impact: This could lead to confusion for developers reading the code, as it suggests that there is a specific error condition that can occur, but it is not actually being checked for or handled anywhere in the contract.

Recommended Mitigation: Change the order of the arguments in the event definition or the event emission to match the correct order.

```
-    emit LiquidityAdded(msg.sender, poolTokensDeposited, wethDeposited
    );
+    emit LiquidityAdded(msg.sender, wethDeposited, poolTokensDeposited
    );
```

[L-2] Default value retuned by `TSwapPool::swapExactInput` results in incorrect value being returned

Description: The `swapExactInput` is expected to return the amount of output tokens received after the swap, but it currently returns a default value of 0. However, while it declares the named returned value `output`, it is never assigned a value, not 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_Always_Returns_Zero() external {
        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 actualAmount = pool.swapExactInput(poolToken, 10e18,
            weth, expected, uint64(block.timestamp));
        vm.stopPrank();

        assert(weth.balanceOf(user) >= expected);
        assertEq(actualAmount, 0, "SwapExactInput should always return
            0");
    }
```

Recommended Mitigation:

```
function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
```

```diff
            uint256 minOutputAmount,
            uint64 deadline
        )
            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);
+             outputAmount = getOutputAmountBasedOnInput(inputAmount,
    inputReserves, outputReserves);

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

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

Informational

[I-1] Unused custom error `PoolFactory::PoolFactory__PoolDoesNotExist` in `PoolFactory` contract

Description: The custom error `PoolFactory__PoolDoesNotExist` is defined in the `PoolFactory` contract but is never used.

Impact: This could lead to confusion for developers reading the code, as it suggests that there is a specific error condition that can occur, but it is not actually being checked for or handled anywhere in the contract.

Recommended Mitigation: Remove the unused error or implement checks for the error condition in the contract.

```diff
-         error PoolFactory__PoolDoesNotExist();
```

[I-2] `wethToken` parameter in the constructor of the `PoolFactory` contract is not checked for zero address

Description: The `wethToken` parameter in the constructor is not validated to ensure it is not a zero address.

Impact: If a zero address is passed as the WETH token address, it could lead to unexpected behavior or vulnerabilities in the contract.

Recommended Mitigation: Add a check in the constructor to ensure that the `wethToken` address is not the zero address.

```
+         require(wethToken != address(0), "WETH token address cannot be zero");
```

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

Description: The `liquidityTokenSymbol` variable is constructed using the `name()` function of the ERC20 token, but it should use the `symbol()` function instead to create a proper token symbol.

Impact: Using the token name instead of the symbol could lead to incorrect or unexpected token symbols being generated for the liquidity tokens.

Recommended Mitigation: Update the `liquidityTokenSymbol` variable to use the `symbol()` function of the ERC20 token instead of the `name()` function.

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

[I-4] `deadline` parameter in `TSwapPool::deposit` function is not used

Description: The `deadline` parameter in the `deposit` function of the `TSwapPool` contract is not utilized within the function body.

Impact: Having unused parameters can lead to confusion and may indicate incomplete functionality. It could also potentially waste gas if the parameter is not needed.

Recommended Mitigation: Implement the logic in `deposit` function to utilize `deadline` parameter effectively.

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

[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.

```
-           event PoolCreated(address tokenAddress, address poolAddress);
+           event PoolCreated(address indexed tokenAddress, address indexed poolAddress);
```

```
-           event LiquidityAdded(address indexed liquidityProvider, uint256 wethDeposited, uint256 poolTokensDeposited);
+           event LiquidityAdded(address indexed liquidityProvider, uint256 indexed wethDeposited, uint256 indexed poolTokensDeposited);
```

```
-           event LiquidityRemoved(address indexed liquidityProvider, uint256 wethWithdrawn, uint256 poolTokensWithdrawn);
+           event LiquidityRemoved(address indexed liquidityProvider, uint256 indexed wethWithdrawn, uint256 indexed poolTokensWithdrawn);
```

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

4 Found Instances

- Found in src/PoolFactory.sol Line: 35

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

- Found in src/TSwapPool.sol Line: 43

```
        event LiquidityAdded(address indexed liquidityProvider, uint256
            wethDeposited, uint256 poolTokensDeposited);
```

- Found in src/TSwapPool.sol Line: 44

```
        event LiquidityRemoved(address indexed liquidityProvider,
           uint256 wethWithdrawn, uint256 poolTokensWithdrawn);
```

- Found in src/TSwapPool.sol Line: 45

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

[I-6] `TSwapPool::TSwapPool__WethDepositAmountTooLow` in this event, `TSwapPool::MINIMUM_WETH_LIQUIDITY` is not required to be emitted as this a constant value

Description: The event `TSwapPool::TSwapPool__WethDepositAmountTooLow` emits the constant value `TSwapPool::MINIMUM_WETH_LIQUIDITY`, which is unnecessary.

Impact: Unnecessary emissions can lead to increased gas costs and cluttered event logs, making it harder for off-chain tools to parse relevant information.

Recommended Mitigation: Remove the emission of `TSwapPool::MINIMUM_WETH_LIQUIDITY` from the event `TSwapPool::TSwapPool__WethDepositAmountTooLow`.

```
-       error TSwapPool__WethDepositAmountTooLow(uint256
    minimumWethDeposit, uint256 wethToDeposit);
+       error TSwapPool__WethDepositAmountTooLow(uint256 wethToDeposit
    );
```

```
-       revert TSwapPool__WethDepositAmountTooLow(
    MINIMUM_WETH_LIQUIDITY, wethToDeposit);
+       revert TSwapPool__WethDepositAmountTooLow(wethToDeposit);
```

[I-7] `TSwapPool::deposit` function is not following CEI for the first-time liquidity deposit

Description: The `TSwapPool::deposit` function does not follow the Checks-Effects-Interactions (CEI) pattern for the first-time liquidity deposit. Specifically, it performs state changes (effects) after interations, which can lead to unexpected behavior and vulnerabilities.

Impact: This can lead to reentrancy attacks or other unexpected behaviors, especially if the contract interacts with external contracts or tokens.

Recommended Mitigation: Ensure that the function follows the CEI pattern by performing all checks first, then making state changes, and finally interacting with external contracts or tokens.

```
        else {
+       liquidityTokensToMint = wethToDeposit;
        _addLiquidityMintAndTransfer(wethToDeposit,
            maximumPoolTokensToDeposit, wethToDeposit);
-       liquidityTokensToMint = wethToDeposit;
        }
```

[I-8] Usage of magic numbers such as 997 and 1000 is not recommended

Description: Magic numbers are hard-coded values that appear in code without explanation. They can make code difficult to understand and maintain.

```
        uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 denominator = (inputReserves * 1000) +
            inputAmountMinusFee;
```

Impact: Using magic numbers can lead to confusion and errors, as the meaning of these numbers is not immediately clear. It can also make future modifications more difficult.

Recommended Mitigation: Define constants for these magic numbers to improve code readability and maintainability.

```
+       uint256 constant FEE_DENOMINATOR = 1000;
+       uint256 constant FEE_MULTIPLIER = 997;
        uint256 inputAmountMinusFee = inputAmount * FEE_MULTIPLIER;
        uint256 denominator = (inputReserves * FEE_DENOMINATOR) +
            inputAmountMinusFee;
```

[I-9] Natspec is not defined for the function `TSwapPool::swapExactInput`

Description: The function `TSwapPool::swapExactInput` does not have Natspec comments defined, which are used to provide documentation for functions, parameters, and return values.

Impact: Without Natspec comments, it is difficult for developers to understand the purpose and usage of the function, which can lead to misuse or errors.

Recommended Mitigation: Add Natspec comments to the function to provide clear documentation for its purpose, parameters, and return values.

```
+   /// @notice Swaps an exact amount of input tokens for as many output tokens as possible
+   /// @param inputToken The address of the input token
+   /// @param inputAmount The exact amount of input tokens to swap
+   /// @param outputToken The address of the output token
+   /// @param minOutputAmount The minimum amount of output tokens to receive
+   /// @param deadline The deadline for the transaction to be completed by
+   /// @return amountOut The amount of output tokens received
        function swapExactInput(
                IERC20 inputToken,
                uint256 inputAmount,
                IERC20 outputToken,
                uint256 minOutputAmount,
                uint64 deadline
        ) external returns (uint256 amountOut) {
                // Swap logic here
        }
```

[I-10] `TSwapPool::swapExactOutput` natspec is missing the `@param` tag for `deadline`

Description: The `@param` tag for `deadline` is missing in the Natspec comments for the `TSwapPool::swapExactOutput` function.

Impact: Without the `@param` tag, users may not understand the purpose of the `deadline` parameter, leading to potential misuse or errors.

Recommended Mitigation: Add the `@param` tag for `deadline` in the Natspec comments for the `TSwapPool::swapExactOutput` function.

```
+   /// @param deadline The deadline for the transaction to be completed by
    /// @return amountIn The amount of input tokens used for the swap
    function swapExactOutput(
        IERC20 inputToken,
        uint256 outputAmount,
        IERC20 outputToken,
        uint256 maxInputAmount,
        uint64 deadline
    )
        external
```

```
            revertIfZero(outputAmount)
            revertIfDeadlinePassed(deadline)
            returns (uint256 amountIn)
        {         // Swap logic here
        }
```

Gas

[G-1] `TSwapPool::swapExactInput` and `TSwapPool::totalLiquidityTokenSupply` functions should be marked as `external` instead of `public`

Description: The `TSwapPool::swapExactInput` and `TSwapPool::totalLiquidityTokenSupply` functions are currently marked as `public`, but they should be marked as `external` to optimize gas usage and restrict access to external calls only.

Impact: Marking the function as `external` can reduce gas costs by allowing the Solidity compiler to optimize the function call.

Recommended Mitigation: Change the visibility of the `swapExactInput` function from `public` to `external`.

```
        function swapExactInput(
            IERC20 inputToken,
            uint256 inputAmount,
            IERC20 outputToken,
            uint256 minOutputAmount,
            uint64 deadline
        )
-        public returns (uint256 amountOut) {
+        external returns (uint256 amountOut) {
            // Swap logic here
        }
```

Change the visibility of the `totalLiquidityTokenSupply` function from `public` to `external`
.

```
        function totalLiquidityTokenSupply()
-        public view returns (uint256) {
+        external view returns (uint256) {
            return totalSupply();
        }
```