TSwap (Uniswap V1) Audit Report

Version 1.0

Yoann LHOMME

July 24, 2025

Protocol Audit Report

Yoann LHOMME

July 24, 2025

Prepared by: Yoann Lead Auditors: - Yoann LHOMME

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
- High
- Medium
- Low
- Informational
- Gas

Protocol Summary

This project is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX). T-Swap is known as an Automated Market Maker (AMM) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset. It is similar to Uniswap.

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.

You can think of each `TSwapPool` contract as it's own exchange between exactly 2 assets. The protocol must be able to do that :

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`

Disclaimer

The whole 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

	Impact			
	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
```

- 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.

Issues found

Severity	Number of issues found
High	4
Medium	2

Severity	Number of issues found
Low	2
Info	7
Total	15

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user should deposit given an amount of 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:

```
function testFlawedSwapExactOutput() 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 outputAmount = 10e18;

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        deal(address(poolToken), user, 200e18);


        uint256 userInitialBalance = poolToken.balanceOf(user);
        pool.swapExactOutput(poolToken, weth, outputAmount, uint64(
            block.timestamp));
        uint256 userFinalBalance = poolToken.balanceOf(user);
        uint256 userBalanceChange = userInitialBalance -
            userFinalBalance;
```

```
19          emit log_named_uint("TokenA paid (should be ~10-11e18)",
                userBalanceChange);
20          assertGt(userBalanceChange, 80e18); // The fees are way too
                high
21      }
```

Recommended Mitigation:

```
1  function getInputAmountBasedOnOutput(
2          uint256 outputAmount,
3          uint256 inputReserves,
4          uint256 outputReserves
5      )
6          public
7          pure
8          revertIfZero(outputAmount)
9          revertIfZero(outputReserves)
10         returns (uint256 inputAmount)
11     {
12 -       return ((inputReserves * outputAmount) * 10000) / ((
       outputReserves - outputAmount) * 997);
13 +       return ((inputReserves * outputAmount) * 1000) / ((
       outputReserves - outputAmount) * 997);
14     }
```

[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 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 should 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.

Here is the Proof of Concept for that specific case :

```
1  function testSwapExactOutputNoSlippageProtection() public {
2          // 1. Setup : init pool with 1000 USDC / 1 WETH
```

```
3           vm.startPrank(liquidityProvider);
4           weth.approve(address(pool), 1e18);
5           poolToken.approve(address(pool), 1000e18);
6           pool.deposit(1e18, 1e18, 1000e18, uint64(block.timestamp));
7           vm.stopPrank();
8
9           // 2. Simulate that the user have a lot of USDC (ex: 20k) but
                to get 1 WETH ==> normally 1000 USDC but he will pay more
                due to the manipulation
10          deal(address(poolToken), user, 20_000e18);
11
12          uint256 outputAmount = 1e18;
13
14          vm.startPrank(user);
15          poolToken.approve(address(pool), type(uint256).max);
16          uint256 userInitialBalance = poolToken.balanceOf(user);
17          vm.stopPrank();
18
19          // Manipulate the price of WETH by an attacker
20          address attacker = makeAddr("attacker");
21          deal(address(poolToken), attacker, 20_000e18);
22
23          vm.startPrank(attacker);
24          poolToken.approve(address(pool), type(uint256).max);
25
26          // Attacker buy massive amount of WETH (inputToken = USDC)
27          // This increases the price of WETH, making the following swap
                very expensive
28          pool.swapExactInput(poolToken, 10_000e18, weth, 0, uint64(block
                .timestamp));
29          vm.stopPrank();
30
31          // 5. Now that the price has been manipulated, the user
                performs their swap
32          // Add more WETH liquidity to ensure the pool has enough, due
                to the flawed getInputAmountBasedOnOutput() function already
                 reported
33          weth.mint(address(pool), 7e18);
34          vm.startPrank(user);
35          pool.swapExactOutput(poolToken, weth, outputAmount, uint64(
                block.timestamp));
36          uint256 userFinalBalance = poolToken.balanceOf(user);
37          vm.stopPrank();
38
39          uint256 paid = userInitialBalance - userFinalBalance;
40          emit log_named_uint("USDC paid for 1 WETH", paid);
41
42          assertGt(paid, 10_000e18, "User should pay more than 10,000
                USDC for 1 WETH due to manipulation");
43      }
```

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 are 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:

Due to the fact that there is a wrong return in the `SwapExactInput` function, we calculate the balance before and after in the helper function.

```
function simulateCorrectSellPoolTokens(uint256 poolTokenAmount)
    internal returns (uint256) {
    poolToken.approve(address(pool), type(uint256).max);

    uint256 wethBefore = weth.balanceOf(address(this));
    pool.swapExactInput(poolToken, poolTokenAmount, weth, 0, uint64(
        block.timestamp));
    uint256 wethAfter = weth.balanceOf(address(this));

    return wethAfter - wethBefore;
}
```

```
10
11 function testSellPoolToken_Flaw() public {
12     vm.startPrank(liquidityProvider);
13     weth.approve(address(pool), 100e18);
14     poolToken.approve(address(pool), 100e18);
15     pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
16     vm.stopPrank();
17
18     poolToken.mint(address(this), 10e18);
19     uint256 expectedWeth = simulateCorrectSellPoolTokens(10e18);
20
21     emit log_named_uint("Expected WETH (if swapExactInput was used)",
           expectedWeth);
22
23     poolToken.mint(user, 200e18);
24     vm.startPrank(user);
25     poolToken.approve(address(pool), type(uint256).max);
26
27     uint256 actualWeth = pool.sellPoolTokens(10e18);
28     emit log_named_uint("Actual WETH (via sellPoolTokens)", actualWeth)
           ;
29     vm.stopPrank();
30
31     // Assert that the user is receiving more WETH than expected due to
           the flawed logic in sellPoolTokens
32     assertGt(actualWeth, expectedWeth);
33 }
```

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.

[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 following block of code is responsible for the isssue.

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

    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));
```

```
17      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp));
18      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp));
19      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      vm.stopPrank();
24
25      uint256 endingY = weth.balanceOf(address(pool));
26      int256 actualDeltaY = int256(endingY) - int256(startingY);
27      assertEq(actualDeltaY, expectedDeltaY);
28  }
```

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  -    if (swap_count >= SWAP_COUNT_MAX) {
3  -        swap_count = 0;
4  -        outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000
       );
5  -    }
```

Medium

[M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline

Description: The `deposit` function accepts a deadline parameter, which according to the documentation is "The deadline for the transaction to be completed by". However, this parameter is not used in this function. As a consequence, operation that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavorable.

Impact: Transactions could be sent when market conditions are unfavorable to deposit, even when adding a deadline parameter.

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

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

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

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

Description: In `TSwapPool::_swap()` function, the contract assumes that the exact inputAmout specified is received in output. This assumption breaks when tokens with non-standard behaviors are used — such as fee-on-transfer tokens (which deduct a fee on each transfer), rebase tokens (which dynamically adjust balances), or ERC-777 tokens (which can invoke hooks like `tokensReceived` during a transfer). In such cases, the contract receives less than the declared input, but still sends the full outputAmount to the user. This breaks the protocol invariant and can lead to an economic imbalance, draining value from the pool.

Impact: A malicious user can exploit this by using a fee-on-transfer or rebasable token to consistently receive more value from the pool than they provide.

Proof of Concept:

Here is the proof of code with 2 files : - A `feeOnTransferToken` mocks : Use to simulate the fees. - A `FeeOnTransferBreaksInvariant.t.sol` : Use to attest there is a fee on transfer when a swap is executed

First, create the mock :

```
// SPDX-Licence-Identifier: MIT
pragma solidity ^0.8.20;

import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract FeeOnTransferToken is ERC20 {
    constructor() ERC20("FeeToken", "FEE") {
        _mint(msg.sender, 1_000_000 ether);
    }

    function transfer(address to, uint256 amount) public override
       returns (bool) {
        uint256 fee = (amount * 10) / 100; // 10% fee
```

```
13          uint256 amountAfterFee = amount - fee;
14          super.transfer(to, amountAfterFee);
15          return true;
16      }
17
18      function transferFrom(address from, address to, uint256 amount)
            public override returns (bool) {
19          uint256 fee = (amount * 10) / 100; // 10% fee
20          uint256 amountAfterFee = amount - fee;
21          _spendAllowance(from, msg.sender, amount);
22          _transfer(from, to, amountAfterFee);
23          return true;
24      }
25  }
```

Then uses this contract that assert that the fee on transfer break the invariant when a swap is executed :

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity 0.8.20;
3
4   import { Test, console } from "forge-std/Test.sol";
5   import { TSwapPool } from "../../src/TSwapPool.sol";
6   import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
        ;
7   import { FeeOnTransferToken } from "../mocks/FeeOnTransferToken.sol";
8   import { ERC20Mock } from "../mocks/ERC20Mock.sol"; // Add this import
        if you have an ERC20Mock contract
9
10  contract FeeOnTransferBreaksInvariant is Test {
11      TSwapPool pool;
12      FeeOnTransferToken feeToken;
13      ERC20Mock outputToken;
14
15      address user = makeAddr("user");
16
17      function setUp() public {
18          feeToken = new FeeOnTransferToken();
19          outputToken = new ERC20Mock();
20
21          pool = new TSwapPool(address(feeToken), address(outputToken), "
                LP", "LPT");
22
23          feeToken.transfer(user, 100 ether);
24          outputToken.mint(address(this), 100 ether);
25          outputToken.transfer(address(pool), 100 ether);
26      }
27
28      function testFeeOnTransferBreaksInvariant() public {
29          vm.startPrank(user);
30
```

```
31          feeToken.approve(address(pool), 10 ether);
32
33          uint256 userOutputBefore = outputToken.balanceOf(user);
34          pool.swapExactInput(feeToken, 10 ether, outputToken, 10 ether,
                uint64(block.timestamp));
35          uint256 userOutputAfter = outputToken.balanceOf(user);
36
37          // The user receives too much output token due to the fee on
                transfer
38          uint256 received = userOutputAfter - userOutputBefore;
39          assertNotEq(received, 10 ether); // The user is receiving more
                than expected so it is not equal to 10 ether
40
41          // The pool receive less token than expected due to the fee on
                transfer
42          uint256 balanceReceived = feeToken.balanceOf(address(pool));
43          assertLt(balanceReceived, 10 ether);
44          vm.stopPrank();
45      }
46  }
```

Recommended Mitigation:

1. Remove the extra incentive mechanism

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

2. Measure the actual received amount using balance differentials before and after the transferFrom:

```
1  +   uint256 balanceBefore = inputToken.balanceOf(address(this));
2
3      // @written breaks protocol invariant!!!
4              swap_count++;
5              if (swap_count >= SWAP_COUNT_MAX) {
6                  swap_count = 0;
7                  outputToken.safeTransfer(msg.sender, 1
                       _000_000_000_000_000_000);
8              }
9              emit Swap(msg.sender, inputToken, inputAmount, outputToken,
                   outputAmount);
10
11     inputToken.safeTransferFrom(msg.sender, address(this), inputAmount)
           ;
12 +   uint256 balanceAfter = inputToken.balanceOf(address(this));
13 +   uint256 actualReceived = balanceAfter - balanceBefore;
```

3. Enforce a check to ensure that the entire inputAmount was received, and revert otherwise:

```
+   uint256 balanceBefore = inputToken.balanceOf(address(this));

    // @written breaks protocol invariant!!!
            swap_count++;
            if (swap_count >= SWAP_COUNT_MAX) {
                swap_count = 0;
                outputToken.safeTransfer(msg.sender, 1
                    _000_000_000_000_000_000);
            }
            emit Swap(msg.sender, inputToken, inputAmount, outputToken,
                outputAmount);

    inputToken.safeTransferFrom(msg.sender, address(this), inputAmount)
        ;
+   uint256 balanceAfter = inputToken.balanceOf(address(this));
+   uint256 actualReceived = balanceAfter - balanceBefore;

+   if (actualReceived < inputAmount) {
+       revert TSwapPool__FeeOnTransferTokenNotSupported();
+   }
```

Low

[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 and the `wethToDeposit` value should go the second parameter.

Impact: Event emission is incorrect, leading to off-chain functions potentially malfunctioning.

Recommended Mitigation:

```
+       emit LiquidityAdded(msg.sender, wethToDeposit,
    poolTokensToDeposit);
-       emit LiquidityAdded(msg.sender, poolTokensToDeposit,
    wethToDeposit);
```

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

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return value `output` it is never assigned a

value, nor uses an explicit return statement.

Impact: The return value will always be 0, giving incorrect information to the caller.

Proof of Concept:

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

        vm.startPrank(user);
        poolToken.approve(address(pool), 10e18);

        uint256 outputReturned = pool.swapExactInput(poolToken, 10e18,
           weth, 9e18, uint64(block.timestamp));
        assertEq(outputReturned, 0);
    }
```

Recommended Mitigation:

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

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

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

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

Informationals

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

Recommended Mitigation:

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

[I-2] Missing zero address checks in `PoolFactory`

Recommended Mitigation:

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

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

Description: The `liquidityTokenName` is alreadly used correctly, however the `liquidityTokenSymbol` is also associated with the `.name()` instead of the `.symbol()`.

Recommended Mitigation:

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

[I-4] Events is missing `indexed` field

Description: 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.

Recommended Mitigation:

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

[I-5] Constant does not require event emission

Description: The constant `MINIMUM_WETH_LIQUIDITY` is used in the `deposit` function to validate the minimum required amount of WETH for a successful deposit. Constants are known at compile time and are publicly accessible in the contract's bytecode. Emitting an event for a static value that cannot change provides no added benefit and incurs unnecessary gas costs.

Impact: It involves unnecessary gas costs.

Proof of Concept:

```
1 if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
2     revert TSwapPool__WethDepositAmountTooLow(MINIMUM_WETH_LIQUIDITY,
          wethToDeposit);
3 }
```

Recommended Mitigation:

There is no need to used the constant `MINIMUM_WETH_LIQUIDITY` to emit an event. Therefore, it can be removed from the error function.

[I-6] The `TSwapPool::poolTokenReserves` variable is not used in this function, so it can be removed

Description: This line must be removed for the purpose of gas efficiency.

Recommended Mitigation:

```
1 if (totalLiquidityTokenSupply() > 0) {
2             uint256 wethReserves = i_wethToken.balanceOf(address(this))
                  ;
3 -           uint256 poolTokenReserves = i_poolToken.balanceOf(address(
      this));
```

[I-7] The `TSwapPool::deposit` has an if-else statement where it doesn't follow CEI

Description: It is best pratice to always follow CEI, therefore it would be better if this was before the `_addLiquidityMintAndTransfer` call

Recommended Mitigation:

```
1     if (totalLiquidityTokenSupply() > 0) {
2     } else {
3         // This will be the "initial" funding of the protocol. We are
              starting from blank here!
```

```
4           // We just have them send the tokens in, and we mint liquidity
                tokens based on the weth
5   +       liquidityTokensToMint = wethToDeposit;
6           _addLiquidityMintAndTransfer(wethToDeposit,
                maximumPoolTokensToDeposit, wethToDeposit);
7   -       liquidityTokensToMint = wethToDeposit;
8       }
```