TSwap Protocol Audit Report

Version 1.0

0xl33

December 13, 2024

TSwap Protocol Audit Report

0xl33

December 13, 2024

Prepared by: 0xl33

Table of Contents

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. To understand Uniswap, please watch this video: Uniswap Explained

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	High	H	H/M	M
	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

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

Issues found

Severity	Number of issues found
High	5
Low	4
Gas	2
Informational	6
Total	17

Findings

High

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

Description: The `deposit` function accepts a `deadline` parameter, which according to the documentation is "The deadline for the transaction to be completed by". However, this parameter is never used. As a consequence, operations 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 completed when market conditions are unfavorable to deposit, even when inputting a deadline value.

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)
    {
```

[H-2] Business logic error in `TSwapPool::getInputAmountBasedOnOutput` function, which leads to an overcharged return value

Description: The `TSwapPool::getInputAmountBasedOnOutput` function includes a logic error where an incorrect multiplier (10000) is applied in the numerator. This multiplier is too large by a factor of 10, resulting in an `inputAmount` that is significantly inflated. This leads to users being overcharged when attempting to swap tokens when calling the `TSwapPool::swapExactOutput` function, causing them to provide 10 times more input than intended.

Impact: This issue has a few effects:

1. Users would lose funds by overpaying for their desired output due to the inflated `inputAmount`.
2. The `TSwapPool::getInputAmountBasedOnOutput` function is called in `TSwapPool::swapExactOutput`, which is one of the main ways users would interract with the protocol. This would cause significant financial discrepancies and breaking the intended functionality of the swap mechanism.
3. If deployed, this issue can lead to users abandoning the protocol due to incorrect and unfair calculations.

Proof of Concept:

The problematic line in question:

```
    return ((inputReserves * outputAmount) * 10000) / ((outputReserves
        - outputAmount) * 997);
```

Example case:

We call the `getInputAmountBasedOnOutput` function with these inputs:

outputAmount = 100 (desired tokens) inputReserves = 100 (ETH in pool) outputReserves = 10,000 (tokens in pool)

Expected Calculation:

((100 * 100) * 1000) / ((10000 − 100) * 997) = 10000000 / 9870300 = 1.013 ETH (including fees)

Actual Calculation (with multiplier error):

((100 * 100) * 10000) / ((10000 − 100) * 997) = 100000000 / 9870300 = 10.13 ETH (10x overcharged)

When we compare the actual and expected calculations we clearly see that the actual returned value will be 10 times bigger than the expected value.

Recommended Mitigation: Update the multiplier in the numerator to the correct value (1000) to align with the intended fee structure and ensure proper input-output calculations. Even better would be to declare two `constant` variables in the `state variables` section, and use these variables instead of "magic" numbers. This way, it's much less likely to run into an issue like this in the future.

Example fix:

```
    IERC20 private immutable i_wethToken;
    IERC20 private immutable i_poolToken;
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
    uint256 private swap_count = 0;
    uint256 private constant SWAP_COUNT_MAX = 10;
+   uint256 private constant FEE_MULTIPLIER = 997;
+   uint256 private constant FEE_DIVISOR = 1000;
```

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        return
-           ((inputReserves * outputAmount) * 10000) /
-           ((outputReserves - outputAmount) * 997);
+           ((inputReserves * outputAmount) * FEE_DIVISOR) /
+           ((outputReserves - outputAmount) * FEE_MULTIPLIER);
    }
```

```
+   function getFeeMultiplier() external view returns (uint256) {
+       return FEE_MULTIPLIER;
+   }

+   function getFeeDivisor() external view returns (uint256) {
+       return FEE_DIVISOR;
```

```
+    }
```

[H-3] Missing `maxInputAmount` Parameter in `TSwapPool::swapExactOutput` Function, which leads to the contract not being constrained by a maximum input limit, resulting in users being overcharged for their desired output.

Description: The `TSwapPool::swapExactOutput` function lacks a `maxInputAmount` parameter, which allows users to specify the maximum amount of input tokens they are willing to spend for the desired output amount. Without this parameter, the contract can take an arbitrary amount of input tokens from the user, leading to a slippage exploit where users may overpay significantly due to unfavorable reserve ratios or unexpected changes in pool conditions.

Impact: This issue makes the callers of the `TSwapPool::swapExactOutput` function use unlimited slippage, which means that every time someone calls this function, they will get sandwiched by MEV bots, overpaying for the specified output by a lot.

Proof of Concept:

Exploit Scenario:

1. A user wants to swap 1 WETH for 100 pool tokens.
2. The reserves change between the user initiating the transaction and it being executed (e.g., due to another large swap), significantly increasing the required `inputAmount`.
3. Without a `maxInputAmount` parameter, the function takes 10 WETH from the user as the `inputAmount`, resulting in significant overpayment.

Recommended Mitigation: Add a `maxInputAmount` parameter to the function, and enforce a check to ensure that the calculated `inputAmount` does not exceed the user's specified maximum. If the condition is violated, the transaction should revert.

```
contract TSwapPool is ERC20 {
+    error TSwapPool__InputAmountTooHigh(uint256 inputAmount, uint256
    maxInputAmount);
```

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    uint256 outputAmount,
+   uint256 maxInputAmount,
    uint64 deadline
)
    public
    revertIfZero(outputAmount)
    revertIfDeadlinePassed(deadline)
```

```
11      returns (uint256 inputAmount)
12  {
13      uint256 inputReserves = inputToken.balanceOf(address(this));
14      uint256 outputReserves = outputToken.balanceOf(address(this));
15
16      inputAmount = getInputAmountBasedOnOutput(
17          outputAmount,
18          inputReserves,
19          outputReserves
20      );
21
22  +   if (inputAmount > maxInputAmount) revert TSwapPool__InputTooHigh(
        inputAmount, maxInputAmount);
23
24      _swap(inputToken, inputAmount, outputToken, outputAmount);
25  }
```

[H-4] Incorrect Swap Function Call in `TSwapPool::sellPoolTokens` Wrapper Function, which leads to the transaction getting reverted.

Description: The `TSwapPool::sellPoolTokens` function is intended to facilitate users selling pool tokens (`poolTokenAmount`) in exchange for WETH (`wethAmount`). However, the function incorrectly calls `swapExactOutput` instead of `swapExactInput`. In this context, the `poolTokenAmount` is passed as the `outputAmount` parameter to `swapExactOutput`, which causes the function logic to behave incorrectly. The intended logic is to call the `swapExactInput` function since `poolTokenAmount` should represent the input tokens being sold, and the WETH received should be the output.

Impact: The `TSwapPool::sellPoolTokens` function logic contradicts its intended purpose, resulting in the incorrect calculation of output amount and the transaction always failing or reverting, which makes the `TSwapPool::sellPoolTokens` function unusable.

Proof of Concept:

Example scenario:

1. A user calls `TSwapPool::sellPoolTokens` function by inputting 100 pool tokens as `poolTokenAmount`.
2. The curent exchange rate for WETH to pool tokens is 1:100.
3. Inside the function, `swapExactOutput` is called and `poolTokenAmount` is passed as the 3rd parameter.
4. Inside `swapExactOutput` function, `getInputAmountBasedOnOutput` is called, and the same amount of 100 pool tokens is passed as the 1st parameter. Currently there is a business

logic issue, described above in our 2nd high risk finding, in `getInputAmountBasedOnOutput` function, but let's assume this issue is fixed.

5. Based on the calculations inside the `getInputAmountBasedOnOutput` function, the return value would be different, depending on reserves in the pool:
    - if the WETH reserves are equal to the pool token amount (100 == 100), then the EVM will throw a panic error with the error code 0x11, which corresponds to "Panic(uint256)" for division or modulo by zero, and the transaction will revert.
    - if the WETH reserves are bigger than the pool token amount (let's say reserves are equal to 101 WETH and pool token amount is equal to 100), then the returned value will be extremely high, in this case ~ 1 003 009 WETH.
        - in this case, `_swap` function will be called, passing in ~ 1 003 009 WETH as the `inputAmount`, and the function will try to transfer ~ 1 003 009 pool tokens from the caller to the contract, which will most likely fail, due to insufficient balance or a missing approval.
    - if the WETH reserves are smaller than the pool token amount (let's say reserves are equal to 99 WETH and pool token amount is equal to 100), then the return value will be negative, and due to this, the EVM will throw a panic error with the error code 0x11, same as before, and the transaction will revert.

Based on this scenario, we can clearly see that the `TSwapPool::sellPoolTokens` function will always fail or revert, which makes this functioin unusable.

Recommended Mitigation: Replace the call to `swapExactOutput` with `swapExactInput` to ensure that `poolTokenAmount` is treated as the input and WETH is the output.

Example fix:

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

[H-5] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`

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

- `x`: The balance of the pool token
- `y`: The balance of WETH
- `k`: The constant product of the two balances

This means, that whenever the balances change in the protocol, the ratio between the two amounts should remain constant, hence the `k`. However, this is broken due to the extra incentive in the `_swap` function. Meaning that over time the protocol funds will be drained.

The following 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` and run the test:

```
    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);
```

```
13          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
14          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
15          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
16          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
23          int256 startingY = int256(weth.balanceOf(address(pool)));
24          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
25
26          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
27          vm.stopPrank();
28
29          uint256 endingY = weth.balanceOf(address(pool));
30          int256 actualDeltaY = int256(endingY) - int256(startingY);
31          assertEq(actualDeltaY, expectedDeltaY);
32      }
```

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 -         }
```

Low

[L-1] The constructors in `PoolFactory` and `TSwapPool` contracts lack zero address checks, which makes it possible for all pools created by the deployed contract to become inoperable

Impact: If the deployer inputs a zero address as `wethToken` when deploying the `PoolFactory` contract or when a caller of the `PoolFactory::CreatePool` function inputs a zero address as `tokenAddress`, all the pools created by the contract will become useless, because `PoolFactory::CreatePool` function uses the `i_wethToken` and `tokenAddress` variables when creating a new pool, and if the values stored in these variables will be the zero address, new pool will get created, but any interractions with the `i_wethToken` or `tokenAddress` variables inside the pool will always fail, due to the zero address not containing any code.

Recommended Mitigation:

1. Firstly, custom errors should be declared, that will be used in the constructors of the corresponding contracts.
2. Secondly, zero address checks should be added to the constructors in `PoolFactory` and `TSwapPool` contracts.
3. Finally, the deployer should double-check the `wethToken` address they are inputting when deploying the `PoolFactory` contract, and the caller of `PoolFactory::CreatePool` function should double-check the `tokenAddress` they are inputting as the parameter, to prevent human error.

```
    contract PoolFactory {
+       error PoolFactory__wethCannotBeInitializedToZeroAddress();
        error PoolFactory__PoolAlreadyExists(address tokenAddress);
```

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

```
    contract TSwapPool is ERC20 {
+       error TSwapPool__wethCannotBeZeroAddress();
+       error TSwapPool__poolTokenCannotBeZeroAddress();
        error TSwapPool__DeadlineHasPassed(uint64 deadline);
```

```
    constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
```

```
6       ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
7   +       if (wethToken == address(0)) revert
        TSwapPool__wethCannotBeZeroAddress();
8   +       if (poolToken == address(0)) revert
        TSwapPool__poolTokenCannotBeZeroAddress();
9           i_wethToken = IERC20(wethToken);
10          i_poolToken = IERC20(poolToken);
11      }
```

[L-2] The `LiquidityAdded` event is emitted with parameters in the wrong order, which could lead to confusion for external systems or users consuming the event logs.

Description: The `LiquidityAdded` event in `TSwapPool::_addLiquidityMintAndTransfer` function is emitted with 2nd and 3rd parameters in the wrong order when compared to the parameter order in the declaration. This issue might create confusion for external systems or users consuming the event logs, as the actual values do not align with their expected positions in the event's declaration. While this issue is low risk, it may cause some minor complications during debugging or when interacting with third-party tools.

Impact: Potential confusion and misalignment in tools parsing the emitted event data.

Proof of Concept:

This is the order of parameters in the declaration of the event:

```
1       event LiquidityAdded(
2           address indexed liquidityProvider,
3           uint256 wethDeposited,
4           uint256 poolTokensDeposited
5       );
```

And this is the order of parameters when the event is emitted in `TSwapPool::_addLiquidityMintAndTransfe` function:

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

Notice, how in the emit statement, the 2nd and 3rd parameters are switched.

Recommended Mitigation: Update the emit statement to ensure that the parameters match the declared order in the event.

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

[L-3] Unused Function Return Parameter in `TSwapPool::swapExactInput`. which leads to the function always returning 0

Description: The `TSwapPool::swapExactInput` function declares a `returns (uint256 output)` statement, but the output parameter is unused within the function. Instead, the variable `outputAmount` is calculated and used internally for the swap logic. Since the `output` parameter is uninitialized, the function will always return 0, leading to misleading behavior. The parameter should be renamed to `outputAmount` to align with its intended use and ensure the function's return value reflects the actual output of the swap.

Impact:

1. External callers expecting the return value to reflect the swap's output will always receive 0, which could cause confusion or misinterpretation of results.
2. The presence of an unused return parameter reduces code clarity and makes debugging more difficult.

Proof of Concept: The `output` parameter is declared as the return value but is never assigned a value, leading to a default return of 0.

Recommended Mitigation: Rename the unused return parameter `output` to `outputAmount` and assign it the value calculated within the function. This ensures the return value correctly reflects the output of the swap.

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (
-           uint256 output
+           uint256 outputAmount
        )
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

-       uint256 outputAmount = getOutputAmountBasedOnInput(
+       outputAmount = getOutputAmountBasedOnInput(
            inputAmount,
            inputReserves,
            outputReserves
```

```
24          );
25
26          if (outputAmount < minOutputAmount) {
27              revert TSwapPool__OutputTooLow(outputAmount,
                    minOutputAmount);
28          }
29
30          _swap(inputToken, inputAmount, outputToken, outputAmount);
31      }
```

[L-4] Misleading Return Values in `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` Functions, which leads to reduced usability and potential downstream issues

Description: The functions `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` both use the `TSwapPool::getOutputAmountBasedO` function to calculate price values. However, when dividing a smaller number (e.g., 1e18) by a larger number, the result can truncate to 0 due to integer division in Solidity. This leads to misleading return values for these getter functions, as they fail to provide accurate price information when the denominator is significantly larger than the numerator.

Impact: This issue has a few effects:

1. Misleading Outputs: The return values may incorrectly appear as 0, which can mislead users or dependent systems into thinking the price is zero.
2. Reduced Usability: These functions do not fulfill their intended purpose of providing accurate price information, reducing the reliability of the contract's pricing mechanism.
3. Potential Downstream Issues: Systems or interfaces relying on these price values may fail or behave incorrectly due to the misleading output.

Proof of Concept:

The current implementation of getPriceOfOneWethInPoolTokens:

```
1 function getPriceOfOneWethInPoolTokens() external view returns (uint256
      ) {
2     return
3         getOutputAmountBasedOnInput(
4             1e18,
5             i_wethToken.balanceOf(address(this)),
6             i_poolToken.balanceOf(address(this))
7         );
8 }
```

Example scenario:

```
i_wethToken.balanceOf(address(this)) = 1e18,
i_poolToken.balanceOf(address(this)) = 1e20
```

Calculation:

```
getOutputAmountBasedOnInput(1e18, 1e18, 1e20),
Result = (1e18 * 1e18) / (1e20) = 1e36 / 1e20 = 1e16 ≈ 0 (truncated to
    uint256)
```

The same issue applies to `getPriceOfOnePoolTokenInWeth`.

Recommended Mitigation: Modify the functions to return a more precise value by scaling the output to avoid truncation issues. Introduce a multiplier (e.g., `1e18`) to preserve the fractional component of the price.

Example fix:

```
function getPriceOfOneWethInPoolTokens() external view returns (uint256
    ) {
     uint256 wethReserves = i_wethToken.balanceOf(address(this));
     uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
-    return getOutputAmountBasedOnInput(1e18, wethReserves,
    poolTokenReserves);
+    return (1e18 * getOutputAmountBasedOnInput(1e18, wethReserves,
    poolTokenReserves)) / 1e18;
}

function getPriceOfOnePoolTokenInWeth() external view returns (uint256)
     {
     uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
     uint256 wethReserves = i_wethToken.balanceOf(address(this));
-    return getOutputAmountBasedOnInput(1e18, poolTokenReserves,
    wethReserves);
+    return (1e18 * getOutputAmountBasedOnInput(1e18, poolTokenReserves,
     wethReserves)) / 1e18;
}
```

Gas

[G-1] The `poolTokenReserves` local variable inside `TSwapPool::deposit` function is unused, and due to this, gas used when calling this function is unnecessasrily increased

Description: `TSwapPool::deposit` function has a local variable called `poolTokenReserves` which is not used anywhere and is not needed.

Impact: In the `TSwapPool::deposit` function, the `poolTokenReserves` local variable is set to the return value of `i_poolToken.balanceOf(address(this))`, which reads from storage

and consumes a lot of gas every time a user calls the `TSwapPool::deposit` function. Due to this, interacting with the `TSwapPool` contract is more expensive. Additionally, this unnecessary variable increases source code size and contributes to the overall complexity and size of the contract.

Recommended Mitigation:

Remove the following line from the code:

```
-   uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

[G-2] Incorrect Visibility Modifier in `TSwapPool::swapExactInput` Function, which leads to gas inefficiency

Description: The `TSwapPool::swapExactInput` function is marked as **`public`**, which allows it to be called both internally and externally. However, the function is not used internally within the contract, making the **`public`** visibility unnecessary. Using the `external` visibility modifier would improve gas efficiency and better reflect the intended usage of the function, as external functions are optimized for calls originating from outside the contract.

Impact:

1. Public functions are less gas-efficient for external calls when compared to external functions, because public functions copy calldata into memory, which uses more gas.
2. The **`public`** visibility may confuse developers into believing the function is meant to be used internally, leading to potential misuse or misunderstanding of the contract's design.

Recommended Mitigation: Update the visibility of the `TSwapPool::swapExactInput` function from **`public`** to `external` to improve gas efficiency and code clarity.

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
-       public
+       external
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (
            uint256 output
        )
```

Informational

[I-1] In `PoolFactory` contract there is a declared custom error, which is not used anywhere, and this increases source code size and creates confusion when reviewing the code.

Description: The `PoolFactory::PoolFactory__PoolDoesNotExist` custom error is declared, but unused.

Impact: This issue has a few effects:

1. The error is part of the Solidity source code, so it contributes to the overall complexity and size of the contract.
2. Auditors may need to investigate why the custom error exists and whether it has a legitimate purpose.
3. If other developers (or even you in the future) read the contract, they may wonder why the error is there if it's never used.
4. It adds unnecessary noise when performing static analysis, potentially triggering warnings for unused declarations.

Recommended Mitigation:

Remove the following line in `PoolFactory` contract:

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

[I-2] In `PoolFactory` and `TSwapPool` contracts there are events, which are missing indexed fields, and due to this, these events will be harder to query and filter programmatically

Description: The `PoolFactory::PoolCreated`, `TSwapPool::LiquidityAdded`, `TSwapPool::LiquidityRemoved`, `TSwapPool::Swap` events have fewer indexed parameters than they should, making it less efficient to filter and query on-chain logs.

Impact: This issue affects usability and efficiency for developers or users relying on event logs for monitoring or integration.

Recommended Mitigation:

Add the `indexed` keyword to fields that are expected to be queried frequently:

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

```
event LiquidityAdded(
        address indexed liquidityProvider,
-       uint256 wethDeposited,
+       uint256 indexed wethDeposited,
-       uint256 poolTokensDeposited
+       uint256 indexed poolTokensDeposited
    );
    event LiquidityRemoved(
        address indexed liquidityProvider,
-       uint256 wethWithdrawn,
+       uint256 indexed wethWithdrawn,
-       uint256 poolTokensWithdrawn
+       uint256 indexed poolTokensWithdrawn
    );
    event Swap(
        address indexed swapper,
-       IERC20 tokenIn,
+       IERC20 indexed tokenIn,
        uint256 amountTokenIn,
-       IERC20 tokenOut,
+       IERC20 indexed tokenOut,
        uint256 amountTokenOut
    );
```

[I-3] When setting `liquidityTokenSymbol`, `IERC20(tokenAddress).name()` is used, which could be not suitable for symbol generation

Description: The `PoolFactory::createPool` function uses `IERC20(tokenAddress).name()` to concatenate a liquidity token symbol. However, the `name()` function can return a very long string, which might not be suitable for symbol generation. The correct function to use is `IERC20(tokenAddress).symbol()`, which is designed to return a shorter, more appropriate value.

Impact:

1. Increased gas costs due to larger strings.
2. Poor user/developer experience if long names are used instead of short, recognizable symbols.

Proof of Concept:

1. Deploy an ERC20 token contract with a long name (e.g., “Super Ultra Long Token Name”).
2. Call the `PoolFactory::createPool` function that uses `IERC20(tokenAddress).name()` for concatenation.
3. Check the value of `liquidityTokenSymbol` by calling `symbol()` function on the created pool address; it will include the long token name instead of its short symbol.

Recommended Mitigation:

Replace `name()` with `symbol()` in the code to ensure shorter, more appropriate values are concatenated:

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

[I-4] In `TSwapPool` contract there is a custom error that emits an already declared constant variable as one of the parameters, which leads to less concise and efficient error reporting.

Description: The `TSwapPool::TSwapPool__WethDepositAmountTooLow` custom error has a parameter `uint256 minimumWethDeposit`, which takes the constant `MINIMUM_WETH_LIQUIDITY` variable. This is unnecessary because the constant value always remains the same, leading to redundant gas usage and less concise error reporting.

Impact:

1. Increased gas usage for each revert.
2. Redundant information in the error, leading to less concise and efficient error reporting.

Recommended Mitigation: Remove the `uint256 minimumWethDeposit` parameter from the error and instead include only `uint256 wethToDeposit`. Users who see the error can reference the constant `MINIMUM_WETH_LIQUIDITY` variable by calling `TSwapPool::getMinimumWethDepositAmount` function.

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

```
    if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
            revert TSwapPool__WethDepositAmountTooLow(
-               MINIMUM_WETH_LIQUIDITY,
                wethToDeposit
            );
        }
```

[I-5] The naming of `TSwapPool__MaxPoolTokenDepositTooHigh` and `TSwapPool__MinLiquidityTokensToMintTooLow` custom errors is inaccurate and misleading

Description: These 2 errors are used in the `TSwapPool::deposit` function and their naming is misleading. In the case of `TSwapPool__MaxPoolTokenDepositTooHigh`, the name should end with `...TooLow`, in the case of `TSwapPool__MinLiquidityTokensToMintTooLow`, the name should end with `TooHigh`. This is clear when we look at the usage of these errors and what kind of scenario they represent. Let's take a look:

```
    if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
        revert TSwapPool__MaxPoolTokenDepositTooHigh(
            maximumPoolTokensToDeposit,
            poolTokensToDeposit
        );
    }
```

Here it will revert if `maximumPoolTokensToDeposit` is a smaller number than `poolTokensToDeposit`, so it would make sense to name the error something like: `maximum...TooLow`.

```
    if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
        revert TSwapPool__MinLiquidityTokensToMintTooLow(
            minimumLiquidityTokensToMint,
            liquidityTokensToMint
        );
    }
```

Here it will revert if `minimumLiquidityTokensToMint` is a bigger number than `liquidityTokensToMint`, so it would make sense to name the error something like: `minimum...TooHigh`.

Impact: The incorrect naming of these custom errors is inaccurate and might mislead users.

Recommended Mitigation:

Change the names of the custom errors, so they represent the scenarios in which they are used:

```
-   error TSwapPool__MinLiquidityTokensToMintTooLow(
+   error TSwapPool__MinLiquidityTokensToMintTooHigh(
        uint256 minimumLiquidityTokensToMint,
        uint256 liquidityTokensToMint
    );
-   error TSwapPool__WethDepositAmountTooLow(
+   error TSwapPool__WethDepositAmountTooHigh(
        uint256 minimumWethDeposit,
        uint256 wethToDeposit
    );
```

```
    if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
```

```
-         revert TSwapPool__MaxPoolTokenDepositTooHigh(
+         revert TSwapPool__MaxPoolTokenDepositTooLow(
              maximumPoolTokensToDeposit,
              poolTokensToDeposit
          );
      }

      liquidityTokensToMint =
          (wethToDeposit * totalLiquidityTokenSupply()) /
          wethReserves;
      if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
-         revert TSwapPool__MinLiquidityTokensToMintTooLow(
+         revert TSwapPool__MinLiquidityTokensToMintTooHigh(
              minimumLiquidityTokensToMint,
              liquidityTokensToMint
          );
      }
```

[I-6] In `TSwapPool::getOutputAmountBasedOnInput` function, there are "magic" numbers used, which contributes to lack of clarity and makes human error more likely when coding

Description: In `TSwapPool::getOutputAmountBasedOnInput` function, there are "magic" numbers `997` and `1000` used directly in the fee calculation logic. Magic numbers are hard-coded values that lack context, making the code harder to read, maintain, and modify. This approach reduces clarity, as it is not immediately apparent what these numbers represent, and updating them in the future may lead to errors if the same value is used in multiple places.

Impact: This issue has a few effects:

1. Readability: Developers may find it difficult to understand the purpose of the magic numbers without comments or additional context.
2. Maintainability: Changes to these values would require modifications in all places where they are used, increasing the risk of errors.
3. Flexibility: Tying these numbers directly into the logic makes it harder to adapt or reuse the code in scenarios with different fee structures.

Recommended Mitigation: Define the "magic" numbers as `constant` state variables with descriptive names, making the purpose of each number explicit and improving code maintainability.

Example fix:

```
    IERC20 private immutable i_wethToken;
    IERC20 private immutable i_poolToken;
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
```

```
4       uint256 private swap_count = 0;
5       uint256 private constant SWAP_COUNT_MAX = 10;
6   +   uint256 private constant FEE_MULTIPLIER = 997;
7   +   uint256 private constant FEE_DIVISOR = 1000;
```

```
1   -   uint256 inputAmountMinusFee = inputAmount * 997;
2   +   uint256 inputAmountMinusFee = inputAmount * FEE_MULTIPLIER;
3       uint256 numerator = inputAmountMinusFee * outputReserves;
4   -   uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
5   +   uint256 denominator = (inputReserves * FEE_DIVISOR) +
        inputAmountMinusFee;
```

```
1   +   function getFeeMultiplier() external view returns (uint256) {
2   +       return FEE_MULTIPLIER;
3   +   }
4
5   +   function getFeeDivisor() external view returns (uint256) {
6   +       return FEE_DIVISOR;
7   +   }
```

This ensures the numbers are self-explanatory and easier to update if needed in the future.