Protocol Audit Report

Version 1.0

xByteNeo

September 23, 2025

TSwap Audit Report

xByteNeo

September 20, 2025

Prepared by: xByteNeo

Lead Security Researcher:

- @xByteNeo

Table of Contents

Protocol Summary

TSwap is a permissionless decentralized exchange (DEX) protocol that allows users to swap assets directly with each other at a fair price. It operates as an Automated Market Maker (AMM), meaning it does not use a traditional order book, but instead relies on liquidity pools of tokens. Users can provide liquidity to these pools and earn fees from swaps. TSwap is inspired by Uniswap V1 and maintains a constant product invariant to ensure fair and efficient trading between assets.

Disclaimer

The xByteNeo solo auditor 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 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 commit hash

```
e643a8d4c2c802490976b538dd009b351b1c8dda
```

Scope

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

- **Solc version:** 0.8.20
- **Chain(s) to deploy to:** Ethereum

Roles

- **Pool Creator**: Any user who calls `PoolFactory::createPool`
  (a) can create new trading pools for ERC20 tokens
  (b) determines the initial pool configuration
- **Liquidity Provider**: Users who provide liquidity to pools
  (a) can call `TSwapPool::deposit` to add liquidity and receive LP tokens

(b) can call `TSwapPool::withdraw` to remove liquidity and burn LP tokens
(c) earns fees from trading activity proportional to their liquidity share

- **Trader/Swapper**: Users who perform token swaps

  (a) can call `TSwapPool::swapExactInput` for exact input swaps
  (b) can call `TSwapPool::swapExactOutput` for exact output swaps

  (c) can call `TSwapPool::sellPoolTokens` as a convenience wrapper
  (d) may receive bonus tokens every 10 swaps as an incentive

Executive Summary

The T-Swap protocol audit revealed multiple critical security vulnerabilities that pose significant risks to protocol functionality and user funds. The contracts suffer from fundamental issues including division by zero vulnerabilities, invariant violations, incorrect swap logic, and improper fee calculations. These vulnerabilities could lead to transaction failures, protocol malfunction, incorrect pricing, and potential loss of user funds. We strongly recommend addressing all identified issues before deploying to mainnet, with particular focus on the high-severity vulnerabilities that directly threaten core protocol mechanics and user assets.

Issues found

Severity	Number of issues found
High	6
Low	2
Informational	11
Gas	1
Total	20

During the audit, a total of 20 issues were identified:

- **6 High severity issues:** division by zero vulnerabilities that could cause transaction failures, incorrect swap logic that breaks core protocol functionality, invariant violations from free token distribution, and incorrect fee calculations that overcharge users.

- **2 Low severity issues:** incorrect event emission parameters and missing return values that affect protocol integration and user experience.
- **11 Informational issues:** code quality improvements including missing input validation, unused code, improper function visibility, and documentation gaps.
- **1 Gas issue:** inefficient parameter usage that increases gas costs.

The High severity issues pose immediate threats to protocol functionality and user funds, with division by zero vulnerabilities and invariant violations being particularly critical as they could cause transaction failures and break the core AMM mechanics. The Low severity issues affect user experience and integration capabilities, while the Informational and Gas issues represent opportunities for code quality improvements.

Findings

High

[H-1] `TSwapPool::deposit` deadline is not used

Description:

The `deadline` parameter is not used in the `deposit` function, causing the protocol to not be able to handle deadline validation. This means that transactions submitted with a deadline will not be rejected even if they are processed after the specified deadline.

Impact:

Users who submit transactions with a deadline expecting them to fail when it reaches the limit will not happen. This severely breaks the protocol's functionality and user trust, as it is a critical security feature that prevents stale transactions from being executed at unfavorable prices. Users may experience unexpected financial losses when their transactions are executed after market conditions have changed significantly.

Recommended Mitigation:

Add `revertIfDeadlinePassed` modifier to the `deposit` function validating the deadline.

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

[H-2] Division by zero in `getPoolTokensToDepositBasedOnWeth` when pool has no WETH reserves

Description:

The `getPoolTokensToDepositBasedOnWeth` function performs division by `wethReserves` without checking by zero. It causes a division by zero exception when the pool has no WETH tokens, which can happen immediately after deployment before any liquidity is added.

Impact:

The function will revert with a division by zero error, completely breaking the functionality, that could occur: - immediately after deployment - if the pool got empty for any reason

The function has no access control and is public, making it allowed to happen in any request made by anyone in the situations mentioned above.

Proof of Concept:

```
// wethReserves zero due to pool being empty
return (wethToDeposit * poolTokenReserves) / wethReserves/*/0/ */;


// both wethReserves and poolTokenReserves are zero due to first
    initialization
return (wethToDeposit * poolTokenReserves/*/0/ */) / wethReserves/*/0/
    */;
```

Recommended Mitigation:

Add a check for zero reserves before performing the division:

```
    function getPoolTokensToDepositBasedOnWeth(
        uint256 wethToDeposit
    ) public view returns (uint256) {
        uint256 poolTokenReserves = i_poolToken.balanceOf(address(this)
            );
        uint256 wethReserves = i_wethToken.balanceOf(address(this));
+
+       if (wethReserves == 0) {
+           revert TSwapPool__PoolHasNoWethReserves();
+       }
+
        return (wethToDeposit * poolTokenReserves) / wethReserves;
    }
```

[H-3] `TSwapPool::sellPoolTokens` uses wrong swap function with incorrect logic

Description:

The `sellPoolTokens` function is implemented incorrectly by calling `swapExactOutput` instead of `swapExactInput`. The function is supposed to sell a specific amount of pool tokens and receive WETH, but it's actually trying to get exactly `poolTokenAmount` of WETH by calculating how many pool tokens are needed. This completely reverses the intended functionality.

Impact:

The function does the opposite of what its name and purpose suggest. Users calling this function expecting to sell pool tokens will get unexpected behavior where the function calculates how many pool tokens are needed to get a specific amount of WETH. Creating severe user confusion, potential financial losses, and making the function unusable for its purpose.

Recommended Mitigation:

Consider changing `sellPoolTokens` to use `swapExactInput` instead of `swapExactOutput`, ensuring it sells the specified amount of pool tokens for at least a minimum amount of WETH, which matches the intended behavior described by the function name and purpose.

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

In addition we also recommend addind a dealine parameter to the function to prevent MEV attacks.

[H-4] Protocol invariant violation due to free token distribution in swap mechanism

Description:

The `_swap` function contains a reward mechanism that distributes free tokens to users after a certain number of swaps, which fundamentally violates the core AMM invariant. The protocol maintains a constant product formula where the product of token reserves should remain constant, but the free token distribution breaks this mathematical relationship.

The problematic code section:

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

Impact:

This creates a critical vulnerability where users can systematically drain protocol funds by repeatedly triggering the reward mechanism. The free token distribution occurs every 10 swaps, allowing attackers to accumulate significant value without providing equivalent liquidity. This breaks the economic model of the AMM and leads to complete fund drainage over time.

Proof of Concept:

An attacker can exploit this by: 1. Performing 10 swaps to trigger the reward mechanism 2. Collecting the free 1 ETH worth of tokens 3. Repeating this process until all protocol funds are exhausted

The constant product invariant `reserveA * reserveB = k` is violated because tokens are removed from the pool without corresponding input, reducing the total value while maintaining the same k value.

Proof Of Code

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

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

    uint256 outputWeth = 1e17;

    vm.startPrank(user);
    poolToken.approve(address(pool), type(uint256).max);
    poolToken.mint(user, 100e18);
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
        timestamp));

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

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

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

Recommended Mitigation: Remove the free token distribution mechanism entirely. If rewards are necessary, implement them through proper fee mechanisms that maintain the invariant:2

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

Alternatively, if rewards must be kept, allocate them from a separate treasury that doesn't affect the pool's reserve calculations.

[H-5] Incorrect fee calculation in `getInputAmountBasedOnOutput` results in excessive user charges

Description:

The `getInputAmountBasedOnOutput` function contains an error in its fee calculation. It is designed to determine the required input amount for an output, accounting for a 0.3% fee (997/1000 ratio). However, the implementation incorrectly multiplies by 10,000 instead of 1,000, causing an overcharge on fees.

```
    return ((inputReserves * outputAmount) * 10_000) / ((outputReserves
        - outputAmount) * 997);
```

Impact:

Users are charged 10 times more fees than intended, resulting in financial losses.

Proof of Concept:

For a swap where: - Input reserves: 1000 tokens - Output reserves: 1000 tokens
- Desired output: 100 tokens - Expected fee: 0.3% (997/1000) - Actual fee charged: 3% (due to 10x multiplier)

Recommended Mitigation:

Correct the fee calculation by changing the multiplier from 10,000 to 1,000:

```diff
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-     return ((inputReserves * outputAmount) * 10_000) / ((
    outputReserves - outputAmount) * 997);
+     return ((inputReserves * outputAmount) * 1_000) / ((outputReserves
     - outputAmount) * 997);
    }
```

[H-6] Invariant break in fee-on-transfer mechanism

Description:

The `_swap` function contains a fee-on-transfer mechanism that breaks the constant product invariant by adding fee to the output token balance.

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

Impact:

From the 10th swap triggers a massive invariant break (~100,000x larger than normal) and severely breaking the protocol economy.

Proof of Concept:

Create a new unit test file in `test/invariant` directory and add the code test below to prove the check invariant break.

Code Test

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

import {Test, console2} from "forge-std/Test.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {PoolFactory} from "../../src/PoolFactory.sol";
import {TSwapPool} from "../../src/TSwapPool.sol";

contract FeeOnTransferInvariantBreakTest is Test {
    ERC20Mock poolToken;
    ERC20Mock weth;
    PoolFactory factory;
    TSwapPool pool;

    uint256 constant STARTING_WETH = 100e18;
    uint256 constant STARTING_POOL_TOKEN = 50e18;
    uint256 constant SWAP_COUNT_MAX = 10;
    uint256 constant FEE_AMOUNT = 1_000_000_000_000_000_000; // 1 ETH
        in wei

    function setUp() public {
        weth = new ERC20Mock();
        poolToken = new ERC20Mock();

        factory = new PoolFactory(address(weth));
        pool = TSwapPool(factory.createPool(address(poolToken)));

        // Mint initial tokens
        poolToken.mint(address(this), STARTING_POOL_TOKEN);
        weth.mint(address(this), STARTING_WETH);

        // Approve pool to spend tokens
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);

        // Initial deposit to establish the pool
        pool.deposit(
            STARTING_WETH,
            STARTING_WETH,
            STARTING_POOL_TOKEN,
            uint64(block.timestamp)
        );
    }

    function getConstantProduct() public view returns (uint256) {
        return
            weth.balanceOf(address(pool)) * poolToken.balanceOf(address
                (pool));
    }

    function abs(int256 x) private pure returns (uint256) {
        return x < 0 ? uint256(-x) : uint256(x);
    }

    function test_automaticInvariantBreakDetection() public {
        address swapper = makeAddr("swapper");
        weth.mint(swapper, 100e18);

        vm.startPrank(swapper);
        weth.approve(address(pool), type(uint256).max);

        int256[] memory kChanges = new int256[](10);
        uint256 threshold = 1e23;
        int256 maxKChange = 0;
        uint256 maxKChangeSwap = 0;

        for (uint256 i = 0; i < 10; i++) {
            uint256 kBefore = getConstantProduct();

            pool.swapExactInput(
                weth,
                1e15,
                poolToken,
                0,
                uint64(block.timestamp + 1)
            );

            uint256 kAfter = getConstantProduct();
            int256 kChange = int256(kAfter) - int256(kBefore);
            kChanges[i] = kChange;

            uint256 absKChange = abs(kChange);
            if (absKChange > abs(maxKChange)) {
                maxKChange = kChange;
                maxKChangeSwap = i + 1;
            }
        }

        vm.stopPrank();

        assertTrue(maxKChangeSwap == 10, "Break should be at swap 10");
        assertTrue(
            abs(maxKChange) > threshold,
            "Break should exceed threshold"
        );
        }
}
```

Recommended Mitigation:

Remove the fee mechanism entirely as it fundamentally breaks the AMM invariant or design a new solution that does not compromise the invariant. We encorage the fix of this issue as soon as possible as it totally breaks the protocol economy if deployed in production.

Low

[L-1] `TSwapPool::_addLiquidityMintAndTransfer` emits event with wrong parameters

Description:

The `LiquidityAdded` event emits the parameters in the wrong order.

Impact:

Emitting the event with parameters in the wrong order causes external tools and users to misinterpret the event data. This can interfere with the usability of the protocol, as analytics, monitoring, and integrations may display incorrect information or fail to function as intended.

Recommended Mitigation:

```
    function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
    ) private {
        _mint(msg.sender, liquidityTokensToMint);
+       emit LiquidityAdded(msg.sender, wethToDeposit, poolTokensToDeposit);
-       emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);

        // Interactions
        i_wethToken.safeTransferFrom(msg.sender, address(this), wethToDeposit);
        i_poolToken.safeTransferFrom(
            msg.sender,
            address(this),
            poolTokensToDeposit
        );
    }
```

[L-2] `swapExactInput` does not return the output amount

Description:

The `swapExactInput` function is declared to return a `uint256 output` but does not actually return any value. This can cause confusion for integrators and may break interfaces that expect a return value.

Impact:

- Integrators will not receive the output amount from the function call, which is critical for tracking swaps and building on top of the protocol.
- May cause issues with tooling and smart contract integrations that expect a return value.

Recommended Mitigation:

Return the `outputAmount` from the function.

```
        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
            );

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

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

Informational

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

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

[I-2] Constructor lacks zero address check

The constructor for `PoolFactory` does not check whether the `wethToken` address is the zero address. This could lead to deployment of a pool with invalid token addresses breaking the core functionality.

Recommendation:

Add zero address check for `wethToken` in the constructor.

```
    constructor(address wethToken) {
+         revert(wethToken != address(0), "WETH token cannot be zero
    address");
          i_wethToken = wethToken;
    }
```

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

The `createPool` function in `PoolFactory` currently uses `.name()` to construct the liquidity token symbol:

Recommendation:

Use `.symbol()` instead of `.name()`.

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

[I-4] `TSwapPool::`Swap event should be indexed

Events with more than three parameters should have at least three parameters marked as `indexed` to facilitate efficient filtering and querying in tools such as The Graph or Etherscan.

Recommendation:

Mark `tokenIn` and `tokenOut` as `indexed` in the `Swap` event definition.

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

[I-5] `TSwapPool` constructor lacks zero address checks

The constructor for `TSwapPool` does not check whether the `poolToken` or `wethToken` addresses are the zero address. This could lead to deployment of a pool with invalid token addresses breaking the core functionality.

Recommendation:

Add zero address checks for both `poolToken` and `wethToken` in the constructor.

```
    constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
+       require(poolToken != address(0), "Pool token cannot be zero address");
+       require(wethToken != address(0), "WETH token cannot be zero address");
        i_wethToken = IERC20(wethToken);
        i_poolToken = IERC20(poolToken);
    }
```

[I-6] `TSwapPool::deposit` includes unnecessary information in custom error

The `TSwapPool::deposit` function is including the `MINIMUM_WETH_LIQUIDITY` value in the custom error `TSwapPool__WethDepositAmountTooLow` that is a constant that could be accessed by anyone just by reading the code.

Recommendation:

Remove the `MINIMUM_WETH_LIQUIDITY` value from the custom error parameters.

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

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

[I-7] `TSwapPool::deposit` prevent for reentrancy attacks

The `TSwapPool::deposit` function is not following the CEI pattern to prevent the reentrancy attack on **`else`** statement for `_addLiquidityMintAndTransfer` step.

Recommendation:

Move the `liquidityTokensToMint` assignment to the top of the **`else`** statement.

```
+   liquidityTokensToMint = wethToDeposit;

    // This will be the "initial" funding of the protocol. We are
       starting from blank here!
    // We just have them send the tokens in, and we mint liquidity
       tokens based on the weth
    _addLiquidityMintAndTransfer(
        wethToDeposit,
        maximumPoolTokensToDeposit,
        wethToDeposit
    );

-   liquidityTokensToMint = wethToDeposit;
```

[I-8] `TSwapPool::getInputAmountBasedOnOutput` using magic numbers

The `TSwapPool::getInputAmountBasedOnOutput` function is using magic numbers for the scaling factor and the fee adjusted multiplier. It could be difficult to understand the code if the numbers are not explained at first hand.

Recommendation:

Define and replace the magic numbers with the constants `SCALING_FACTOR` and `FEE_ADJUSTED_MULTIPLIER` .

```
+     uint256 public constant SCALING_FACTOR = 10000;
+     uint256 public constant FEE_ADJUSTED_MULTIPLIER = 997;

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

[I-9] `TSwapPool::swapExactInput` lacks proper NatSpec documentation

The `swapExactInput` function is missing comprehensive NatSpec documentation, which is essential for understanding the function's purpose, parameters, return values, and potential side effects.

Recommendation:

```
+     /**
+      * @notice Swaps an exact amount of input tokens for as many
    output tokens as possible.
+      * @dev The function will revert if the output amount is less than
     the minimum specified or if the deadline has passed.
+      * @param inputToken The ERC20 token to swap from (e.g., DAI).
+      * @param inputAmount The exact amount of input tokens to swap.
+      * @param outputToken The ERC20 token to receive (e.g., WETH).
+      * @param minOutputAmount The minimum acceptable amount of output
    tokens to receive.
+      * @param deadline The timestamp by which the swap must be
    completed.
+      * @return output The actual amount of output tokens received from
     the swap.
+      */
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
```

[I-10] `TSwapPool::swapExactInput` should be `external` instead of `public`

Marking it as `external` can save gas and better reflects its intended usage.

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

[I-11] `TSwapPool::totalLiquidityTokenSupply` should be `external` instead of `public`

Marking it as `external` can save gas and better reflects its intended usage.

```
         /// @notice a more verbose way of getting the total supply of
             liquidity tokens
-        function totalLiquidityTokenSupply() public view returns (uint256)
        {
+        function totalLiquidityTokenSupply() external view returns (
      uint256) {
            return totalSupply();
        }
```

Gas

[G-1] `TSwapPool::deposit` using `wethToDeposit` instead of `poolTokensToDeposit`

Remove the unused `poolTokenReserves` variable to reduce gas usage per transaction as per it is being ignored across the function.

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