TSwap Potocol Audit Report

Version 1.0

What Fate

August 22, 2025

Protocol Audit Report

What Fate

August 22, 2025

Lead Auditors: - What Fate

Table of Contents

- Medium Severity
  - [M-1] `TSwapPool::deposit` does not enforce `deadline`, risking post-deadline execution
- Low Severity
  - [L-1] Liquidity token symbol uses `.name()` instead of `.symbol()` in `PoolFactory::createPool`
  - [L-2] `TSwapPool::LiquidityAdded` event has parameters out of order
  - [L-3] `TSwapPool::swapExactInput` returns default zero due to uninitialized output variable
- Informational
  - [I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is unused
  - [I-2] Missing zero-address validation in `PoolFactory` constructor
  - [I-3] Missing zero-address validation in `TSwapPool` constructor
  - [I-4] Event `TSwapPool::Swap` has more than 3 parameters and should index key fields
  - [I-5] Missing `deadline` NatSpec in `TSwapPool::swapExactOutput`
  - [I-6] CEI pattern not followed in `TSwapPool::deposit`
  - [I-7] Missing NatSpec in `TSwapPool` functions
- Gas Optimization
  - [G-1] `TSwapPool::swapExactInput` can be `external`

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

Disclaimer

The Hash Horizon team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

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

Audit Details

- Commit Hash: 55d1e086ed0917fd055b14f63099c2342eb6b86a

Scope

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

Roles

- Liquidity Providers: Users who have liquidity deposited into the pools. Their shares are represented by the LP ERC20 tokens. They gain a 0.3% fee every time a swap is made.
- Users: Users who want to swap tokens.

Executive Summary

The audit of `TSwapPool` and `PoolFactory` revealed critical vulnerabilities affecting fund safety and protocol integrity. Bonus token distributions can break the AMM invariant, allowing attackers to drain liquidity, while fee miscalculations and missing slippage protections expose users to financial loss. Other issues include incorrect function calls, unused parameters, and missing documentation, which reduce usability and maintainability. Addressing these findings is essential before mainnet deployment to ensure security, predictable behavior, and user trust.

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	3
Info	7
Gas	1
Total	16

Findings

High Severity

[H-1] Protocol invariant broken due to bonus token giveaway in `TSwapPool::_swap`

Description: Giving bonus tokens from the pool breaks the invariant `x * y = k`, because the bonus tokens distributed to the user reduce the pool reserves outside of the swap formula, leading to a mismatch between actual balances and the expected protocol state. This undermines the AMM pricing mechanism, as the pool can no longer guarantee that its reserves reflect the invariant relationship.

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

Impact: A malicious user can repeatedly perform swaps with minimal input amounts. Every 10th swap grants the attacker bonus tokens directly from the pool reserves. Over time, this mechanism enables the attacker to drain the pool completely, causing liquidity providers to lose their deposited funds. The attack is highly cost-efficient, since the attacker only needs to cover the swap fees, while the bonus tokens received far exceed the amounts spent.

Proof of Concept:

1. Liquidity provider deposits equal amounts of poolToken and WETH into the pool using `TSwapPool::deposit`.
2. Malicious user performs repeated swaps, triggering the bonus token mechanism. Every 10th swap grants bonus tokens directly from the pool reserves.
3. The repeated bonus payouts allow the attacker to drain the pool, violating the `x * y = k` invariant.
4. Post-conditions:

   - User balance increased beyond normal swap gains.
   - Pool reserves decreased below expected invariant.

Proof of Code

Code

```
    function test_swapExactInput_InvariantViolationByBonus() public {
        uint256 SWAP_AMOUNT = 1e17;
        uint256 TOKEN_AMOUNT = 10e18;

        // --- Step 1: Deposit liquidity ---
        // Liquidity provider deposits equal amounts of poolToken and
            WETH into the pool
        // This sets up the initial pool reserves for swaps
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), TOKEN_AMOUNT);
        poolToken.approve(address(pool), TOKEN_AMOUNT);
        pool.deposit(TOKEN_AMOUNT, TOKEN_AMOUNT, TOKEN_AMOUNT, uint64(
            block.timestamp));
        vm.stopPrank();

        // --- Step 2: Perform repeated swaps to trigger bonus tokens
            ---
        // The attacker executes many swaps in order to trigger the
            bonus token mechanism
        // Every 10th swap grants bonus tokens directly from the pool
            reserves
        vm.startPrank(user);

        uint256 startingUserPoolTokenBalance = poolToken.balanceOf(user
            );
        uint256 startingUserWethBalance = weth.balanceOf(user);

        uint256 startingPoolPoolTokenBalance = poolToken.balanceOf(
            address(pool));
        uint256 startingPoolWethBalance = weth.balanceOf(address(pool))
            ;

        // Approve large amounts for repeated swapping
        weth.approve(address(pool), 1000e18);
        poolToken.approve(address(pool), 1000e18);

        // Execute multiple swaps to trigger bonus payouts repeatedly
        for (uint256 i = 0; i < 60; i++) {
            pool.swapExactInput(poolToken, SWAP_AMOUNT, weth, 0, uint64(block.timestamp));
            pool.swapExactInput(weth, SWAP_AMOUNT, poolToken, 0, uint64(block.timestamp));
        }
        // Additional swaps to ensure bonus mechanism triggers more times
        for (uint256 i = 0; i < 10; i++) {
            pool.swapExactInput(poolToken, SWAP_AMOUNT, weth, 0, uint64(block.timestamp));
        }

        vm.stopPrank();

        uint256 endingUserPoolTokenBalance = poolToken.balanceOf(user);
        uint256 endingUserWethBalance = weth.balanceOf(user);

        uint256 endingPoolPoolTokenBalance = poolToken.balanceOf(address(pool));
        uint256 endingPoolWethBalance = weth.balanceOf(address(pool));

        // --- Step 3: Verify pool invariant is violated ---

        // User received extra pool tokens due to the bonus mechanism
        assertLt(startingUserPoolTokenBalance, endingUserPoolTokenBalance);
        // User received extra WETH due to the bonus mechanism
        assertLt(startingUserWethBalance, endingUserWethBalance);
        // Pool's poolToken reserves decreased beyond what the invariant allows
        assertGt(startingPoolPoolTokenBalance, endingPoolPoolTokenBalance);
        // Pool's WETH reserves decreased beyond what the invariant allows
        assertGt(startingPoolWethBalance, endingPoolWethBalance);
    }
```

Recommended Mitigation: Do not distribute bonus tokens directly from the pool, as this breaks the `x * y = k` invariant. Instead, issue bonus tokens from a separate reward reserve contract. Implement strict limits on both the frequency and the amount of bonus distributions. Bonuses should be rare and pre-determined, preventing an attacker from draining the pool through repeated swaps. Additionally, ensure that all bonus mechanisms are tested against the pool invariant to avoid unintended depletion of reserves.

[H-2] Miscalculated fee in `TSwapPool::getInputAmountBasedOnOutput` leads to excessive user charges

Description: The `getInputAmountBasedOnOutput` is incorrectly scales fees by `10_000` instead of `1_000`. This overestimation causes the protocol to require more input tokens than intended, resulting in users being overcharged and losing funds through inflated fees.

As a result, users are forced to deposit more tokens than necessary, effectively being overcharged by the protocol.

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

Impact:

- Users consistently lose funds due to inflated swap costs.
- Protocol collects unintended excess fees, creating accounting mismatches and trust issues.

Proof of Concept:

The following test demonstrates the discrepancy between the correct formula and the current implementation:

Proof of Code

Code

```
    function test_getInputAmountBasedOnOutput_feeMiscalculation()
       public {
        uint256 initialLiquidity = 10e18;
        uint256 desiredOutputAmount = 2e18;

        uint256 feeDenominator = 1000;
        uint256 feePercent = 3; // 0.3%
        uint256 feeMultiplier = feeDenominator - feePercent;

        // --- Arrange: provide initial liquidity to the pool ---
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit(initialLiquidity, initialLiquidity,
            initialLiquidity, uint64(block.timestamp));
        vm.stopPrank();

        uint256 inputReserves = weth.balanceOf(address(pool));
        uint256 outputReserves = poolToken.balanceOf(address(pool));

        // Expected formula with correct fee scaling (1000 basis points
            , fee 0.3%)
        uint256 expectedInputAmount = ((inputReserves *
            desiredOutputAmount) * feeDenominator)
             / ((outputReserves - desiredOutputAmount) * feeMultiplier);

        // Act: call the vulnerable function
        uint256 actualInputAmount = pool.getInputAmountBasedOnOutput(
            desiredOutputAmount, inputReserves, outputReserves);
        // --- Assert: the function takes MORE than it should due to
            miscalculation ---
        assert(actualInputAmount != expectedInputAmount);
    }
```

Recommended Mitigation: Ensure fees are applied using the correct basis point denominator (1,000), consistent with Uniswap-like models.

```
-    return ((inputReserves * outputAmount) * 10_000) / ((outputReserves
     - outputAmount) * 997);
+    return ((inputReserves * outputAmount) * 1_000) / ((outputReserves
    - outputAmount) * 997);
```

[H-3] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

Description: The `swapExactOutput` function does not include any sort of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput`, where the function specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact: If market conditions change before the transaction precesses, the user could get a much worse swap

Proof of Concept:

1. The price of 1 WETH right now is 1,000 USDC
2. User inputs a swapExactOutput looking for 1 WETH

    - inputToken = USDC
    - outputToken = WETH
    - outputAmount = 1
    - 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

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-4] Incorrect function call in `TSwapPool::sellPoolTokens` causes mismatched swap direction and wrong token amounts

Description: The `sellPoolTokens` function is designed to let users redeem their pool tokens (`poolTokenAmount`) for the equivalent amount of WETH. However, the function incorrectly calls `swapExactOutput` instead of `swapExactInput`.

Since users specify the exact number of pool tokens they want to sell (input), the swap should be processed using `swapExactInput`. By using `swapExactOutput`, the logic instead assumes the user is targeting a desired output amount, which misaligns the calculation and results in the wrong amount of tokens being transferred.

Impact:

- Users may receive an incorrect amount of WETH for their pool tokens.
- This breaks the expected behavior of the `sellPoolTokens` function and can cause severe financial loss or complete malfunction of a core feature in the protocol.

- If integrated into a UI, users will consistently experience failed or incorrect swaps, undermining trust in the protocol.

Proof of Concept: A user calls `sellPoolTokens(1e18)`, expecting to receive the equivalent WETH for their pool tokens. Instead, since the function uses `swapExactOutput`, the protocol interprets the input as a desired output, leading to either zero tokens received or an amount that does not match the pool token input.

Recommended Mitigation: Update the `sellPoolTokens` implementation to use `swapExactInput` instead of `swapExactOutput`, since users specify the exact number of pool tokens they want to sell. Additionally, introduce a `minWethToReceive` parameter to provide slippage protection, ensuring users are guaranteed a minimum amount of WETH.

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

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

Medium Severity

[M-1] `TSwapPool::deposit` does not enforce `deadline`, risking post-deadline execution

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: Transaction 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)
    {...}
```

Low Severity

[L-1] Liquidity token symbol uses `.name()` instead of `.symbol()` in `PoolFactory::createPool`

Description: The liquidity token symbol is generated using the token's name instead of its symbol. This can cause confusion in user interfaces, dashboards, and third-party integrations (such as DEXs) that rely on the correct token symbol for identification. Users may see misleading or inconsistent information, potentially leading to mistakes when interacting with the pool.

```
        function createPool(address tokenAddress) external returns (address
            ) {
            if (s_pools[tokenAddress] != address(0)) {
                revert PoolFactory__PoolAlreadyExists(tokenAddress);
            }

            string memory liquidityTokenName = string.concat("T-Swap ",
               IERC20(tokenAddress).name());
-->         string memory liquidityTokenSymbol = string.concat("ts", IERC20
        (tokenAddress).name());

            TSwapPool tPool = new TSwapPool(tokenAddress, i_wethToken,
               liquidityTokenName, liquidityTokenSymbol);
            s_pools[tokenAddress] = address(tPool);
            s_tokens[address(tPool)] = tokenAddress;
            emit PoolCreated(tokenAddress, address(tPool));
            return address(tPool);
        }
```

Impact: Using `.name()` instead of `.symbol()` can confuse users and third-party interfaces, showing misleading token symbols without affecting funds.

Proof of Concept:

1. Deploy a pool using `PoolFactory::createPool` for any ERC20 token.
2. Observe that the liquidity token symbol is incorrectly derived from the token's name:

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

3. On wallets, UIs, or DEXs, the liquidity token displays a misleading symbol, e.g., `"T-Swap"` instead of `"ts"`.
4. Integrations relying on the token symbol may display or categorize it incorrectly.

Recommended Mitigation: Replace the use of `.name()` with `.symbol()` when generating the liquidity token symbol, as shown in the code snippet.

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

[L-2] `TSwapPool::LiquidityAdded` event has parameters out of order

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

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

Proof of Concept:

Recommended Mitigation:

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

[L-3] `TSwapPool::swapExactInput` returns default zero due to uninitialized output variable

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, the declared return variable `output` is never assigned a value nor explicitly returned. As a result, the function always returns `0`, providing incorrect information to callers and misleading external integrations.

Impact: Callers and external contracts relying on the return value receive 0 instead of the actual amount swapped, which can cause incorrect accounting, UI misrepresentation, or unexpected behavior in automated strategies.

Proof of Concept: The following test demonstrates this behavior in practice.

Proof of Code:

Code

```
    function test_swapExactInput_returnsDefaultZero() public {
        uint256 initialLiquidity = 10e18;
        uint256 swapAmount = 4e18;

        // --- Arrange: provide initial liquidity to the pool ---
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit(initialLiquidity, initialLiquidity,
           initialLiquidity, uint64(block.timestamp));
        vm.stopPrank();

        // --- Act: user attempts to swap tokens ---
        vm.startPrank(user);
        weth.approve(address(pool), swapAmount);
        poolToken.approve(address(pool), swapAmount);
        uint256 outputAmount = pool.swapExactInput(weth, swapAmount,
           poolToken, 0, uint64(block.timestamp));
        vm.stopPrank();

        // --- Assert: the returned output is zero due to missing
           assignment ---
        assertEq(outputAmount, 0, "swapExactInput should return 0 when
           output is not assigned");
    }
```

Recommended Mitigation: Assign the calculated output to the return variable to ensure the function returns the correct swapped amount. This prevents misleading results and potential issues in UI and integrations.

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

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

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

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

Informational

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is unused

Description: The error `PoolFactory__PoolDoesNotExist` is declared but never used. Keeping unused errors adds unnecessary complexity and may confuse developers.

Recommended Mitigation:

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

[I-2] Missing zero-address validation in `PoolFactory` constructor

Description: Constructor does not validate `wethToken` against `address(0)`, allowing deployment with an invalid WETH address.

Impact:

- PoolFactory may deploy pools referencing `address(0)`, causing token calls to revert or behave incorrectly.
- Breaks integrations that assume a valid WETH token; potential DoS and costly migration.

Proof of Concept:

1. Deploy `PoolFactory` with `wethToken == address(0)`.
2. Attempt to create or interact with pools — token calls will fail or produce incorrect bookkeeping.

Recommended Mitigation: Validate input in the constructor and provide a clear error message.

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

[I-3] Missing zero-address validation in `TSwapPool` constructor

Description: Constructor does not validate `i_wethToken` and `i_poolToken` against `address(0)`, allowing deployment with invalid token addresses.

Impact:

- Pool functions will fail or behave incorrectly (DoS, broken accounting).
- Unexpected state / invariant violations and costly migration to fix.

Proof of Concept:

1. Deploy `TSwapPool` with `i_wethToken == address(0)` (or `i_poolToken == address(0)`).
2. Calls that interact with the token (e.g., `deposit`, `swap`) will revert or not transfer funds, breaking pool logic.

Recommended Mitigation: Add zero-address checks in the constructor and unit tests.

```
    constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    )
        ERC20(liquidityTokenName, liquidityTokenSymbol)
    {
+       if (wethToken == address(0) || poolToken == address(0)) {
+           revert();
+       }
        i_wethToken = IERC20(wethToken);
        i_poolToken = IERC20(poolToken);
    }
```

[I-4] Event `TSwapPool::Swap` has more than 3 parameters and should index key fields

Description: The `Swap` event has five parameters but only the `swapper` is indexed. Indexing additional key parameters (`tokenIn` and `tokenOut`) improves filtering and efficiency for off-chain listeners.

Recommended Mitigation:

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

[I-5] Missing `deadline` NatSpec in `TSwapPool::swapExactOutput`

Description: `swapExactOutput` accepts a `deadline` parameter but the NatSpec is missing.

```
    /*
     * @notice figures out how much you need to input based on how much
     * output you want to receive.
     *
     * Example: You say "I want 10 output WETH, and my input is DAI"
     * The function will figure out how much DAI you need to input to
        get 10 WETH
     * And then execute the swap
     * @param inputToken ERC20 token to pull from caller
     * @param outputToken ERC20 token to send to caller
     * @param outputAmount The exact amount of tokens to send to caller
-->  *
     */
    function swapExactOutput(
```

Recommended Mitigation: Add a concise `@param` entry describing the parameter and expected behavior.

[I-6] CEI pattern not followed in `TSwapPool::deposit`

Description: In the initial deposit branch, `liquidityTokensToMint` is assigned after calling `_addLiquidityMintAndTransfer`. Following CEI consistently—even for local variables—improves clarity and reduces potential mistakes.

Recommended Mitigation:

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

[I-7] Missing NatSpec in `TSwapPool` functions

Description: `getOutputAmountBasedOnInput`, `getInputAmountBasedOnOutput`, and `swapExactInput` lack NatSpec documentation. Without it, users and integrators cannot easily understand function purpose, inputs, and outputs. Providing NatSpec improves readability, usability, and auditing.

Recommended Mitigation: Add NatSpec for each function.

Gas Optimization

[G-1] `TSwapPool::swapExactInput` can be `external`

Description: `swapExactInput` is not called internally; marking it `external` saves gas and signals intended usage.

Recommended Mitigation:

```
    function swapExactInput(
        ...
    )
-       public
+       external
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 output)
    {
        ...
    }
```