TSwap Protocol Audit Report

Version 1.0

AlexGer

August 28, 2025

TSwap Protocol Audit Report

AlexGer

August 28, 2025

Prepared by: AlexGer Lead Auditors:

- Aleksei Gerasev

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees
    * [H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes user to potentially receive way fewer tokens
    * [H-3] `TSwapPool::sellTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens

    * [H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`
  - Medium
    * [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline
    * [M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol invariant
  - Low
    * [L-1] `TSwapPool::LiquidityAdded` event has parameteres out of order
    * [L-2] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given
  - Info
    * [I-1] Lack of zero address validation
    * [I-2] `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed
    * [I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`
    * [I-4] Event `TSwapPool::Swap` is missing `indexed` fields
    * [I-5] Function `TSwapPool::swapExactInput` should be `external` instead of `public` because it is not used internally
    * [I-6] Missing natspec for the `TSwapPool::swapExactInput` function
    * [I-7] Missing natspec for the `deadline` parameter in the `TSwapPool::swapExactOutput` function
    * [I-8] Using magic numbers instead of constants
    * [I-9] It's better to follow the CEI (Checks, Effects, and Interactions) principle in the else branch of the `TSwap::deposit` function
  - Gas
    * [G-1] Function `TSwapPool::_isUnknown` can be simplified
    * [G-2] Local variable `poolTokenReserves` in the function `TSwapPool::deposit` is not used

Protocol Summary

TSWAP is an constant-product AMM that allows users permissionlessly trade WETH and any other ERC20 token set during deployment. Users can trade without restrictions, just paying a tiny fee in each swapping operation. Fees are earned by liquidity providers, who can deposit and withdraw liquidity at any time.

Disclaimer

The AlexGer 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: e643a8d4c2c802490976b538dd009b351b1c8dda

Scope

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- Tokens:
  - Any ERC20 token

Roles

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

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	9
Gas	2
Total	19

Findings

High

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

Description: The `getInputAmountBasedOnOutput` fucntion is intended to calculate the amount of tokens a user should deposit given an amount of tokens of output tokens. However, the function currently miscalculates the resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Proof of Concept:

Place this code in `TSwapPool.t.sol`

```
    function testCollectFeesWrong() public {
        uint256 wethAmount = 100e18;
        uint256 poolTokenAmount = 100e18;

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), wethAmount);
        poolToken.approve(address(pool), poolTokenAmount);
        pool.deposit({
            wethToDeposit: wethAmount,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: poolTokenAmount,
            deadline: uint64(block.timestamp)
        });

        vm.stopPrank();

        address someUser = makeAddr("someUser");
        uint256 initialUserPoolTokenAmount = 11e18;
        uint256 outputWethUserAmount = 1 ether;
        poolToken.mint(someUser, initialUserPoolTokenAmount);

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

        // in the pool now 1:1, so the user should get 1 WETH and pay
            ~1 poolToken
        pool.swapExactOutput({
            inputToken: poolToken,
            outputToken: weth,
            outputAmount: outputWethUserAmount,
            deadline: uint64(block.timestamp)
        });

        vm.stopPrank();

        uint256 userPoolTokenBalanceAfter = poolToken.balanceOf(
            someUser);
        console.log("userPoolTokenBalanceAfter",
            userPoolTokenBalanceAfter);

        uint256 userSpentPoolToken = initialUserPoolTokenAmount -
            userPoolTokenBalanceAfter;
        console.log("userSpentPoolToken", userSpentPoolToken);
        // user spent more than 10 poolTokens
        assertGt(userSpentPoolToken, 10e18);
    }
```

Recommended Mitigation:

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        return
-           ((inputReserves * outputAmount) * 10000) /
+           ((inputReserves * outputAmount) * 1000) /
            ((outputReserves - outputAmount) * 997);
    }
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes user 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 fucntion specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

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

Proof of Concept:

1. The price of WETH right now is 1,000 USDC
2. User inputs a `swapExactOutput` looking for 1 WETH
    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline = whatever
3. The function does not offer a maxInput amount
4. As the transaction is pending in the mempool, the market changes! And the price moves HUGE -> 1 WETH is now 10,000 USDC. 10x more than the user expected
5. The transaction completes, but the user sent the protocol 10,000 USDC instead of the expected 1,000 USDC

Proof of Code

Place this in `TSwapPool.t.sol`

```
    function testNoSlippageProtection() public {
        uint256 initialLiquidity = 10e18;

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        address swapper = makeAddr("swapper");
        uint256 swapperAmount = 9e18;
        weth.mint(swapper, swapperAmount);

        vm.startPrank(swapper);
        weth.approve(address(pool), type(uint256).max);
        pool.swapExactInput({
            inputToken: weth,
            inputAmount: swapperAmount,
            outputToken: poolToken,
            minOutputAmount: 0,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // ~4.7 pool token after swap
        uint256 poolTokenBalanceAfter = poolToken.balanceOf(swapper);
        console.log("poolTokenBalanceAfter", poolTokenBalanceAfter);

        uint256 userAmount = 100e18;
        weth.mint(user, userAmount);

        vm.startPrank(user);
        weth.approve(address(pool), type(uint256).max);
        // and here we have a huge slippage and the worst course
        uint256 userPaid = pool.swapExactOutput({
            inputToken: weth,
            outputToken: poolToken,
            outputAmount: 1e18,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // user paid much more than the initial liquidity!
        console.log("userPaid for 1 WETH", userPaid);
        assertGt(userPaid, initialLiquidity);
    }
```

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,
        IERC20 outputToken,
        uint256 outputAmount,
        uint64 deadline
    )
        public
        revertIfZero(outputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 inputAmount)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        inputAmount = getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

+       if (inputAmount > maxInputAmount) {
+           revert();
+       }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
    }
```

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

Description: The `sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. Users indicate how many tokens they're willing to sell in the `poolTokenAmount` parameter. However, the function currently miscalculates the swapped amount.

This is due to the fact that the `swapExactOutput` function is called, whereas the `swapExactInput` function is the one that should be called. Because users specify the exact amount of input tokens, not output.

Impact: Users will swap the wrong amount of tokens, which is a severe disruption of protocol functionality.

Proof of Concept:

Proof of Code

Place this in `TSwapPool.t.sol`

```
function testSellPoolTokensSwapsWrongAmount(
            uint256 poolTokenAmount
        ) public {
            uint256 initialLiquidity = 100e30;
            poolTokenAmount = bound(poolTokenAmount, 0, initialLiquidity);
            weth.mint(liquidityProvider, initialLiquidity);
            poolToken.mint(liquidityProvider, initialLiquidity);

            vm.startPrank(liquidityProvider);
            weth.approve(address(pool), initialLiquidity);
            poolToken.approve(address(pool), initialLiquidity);
            pool.deposit({
                wethToDeposit: initialLiquidity,
                minimumLiquidityTokensToMint: 0,
                maximumPoolTokensToDeposit: initialLiquidity,
                deadline: uint64(block.timestamp)
            });
            vm.stopPrank();

            address swapper = makeAddr("swapper");
            poolToken.mint(swapper, poolTokenAmount);

            //try to sell poolToken for WETH with sellPoolTokens
            vm.startPrank(swapper);
            poolToken.approve(address(pool), type(uint256).max);
            // this should revert with any amount because inside
                sellPoolTokens we call swapExactOutput
            vm.expectRevert();
            pool.sellPoolTokens(poolTokenAmount);
            vm.stopPrank();
        }
```

Recommended Mitigation: Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (ie `minWethToReceive` to be passed to `swapExactOutput`)

```
        function sellPoolTokens(
            uint256 poolTokenAmount,
+           uint256 minWethToReceive
        ) external returns (uint256 wethAmount) {
            return
-               swapExactOutput(
+               swapExactInput(
                    i_poolToken,
+                   poolTokenAmount
```

```
                    i_wethToken,
+                   minWethToReceive,
-                   poolTokenAmount,
                    uint64(block.timestamp)
                );
        }
```

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

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

Description: The protocol follows a strict invariant of `x * y = k`. Where:

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

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.

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.

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);
}
```

Proof of Concept:

1. A user swaps 10 times, and collects the extra incentive of `1_000_000_000_000_000_000` tokens
2. That user continues to swap untill all the protocol funds are drained

Proof of Code

Place the following into `TSwapPool.t.sol`

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

        uint256 wethAmount = 1e17;

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);

        for (uint256 i = 1; i <= 9; i++) {
            pool.swapExactOutput(
                poolToken,
                weth,
                wethAmount,
                uint64(block.timestamp)
            );
        }

        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(wethAmount);

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

        vm.stopPrank();

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

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.

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

Medium

[M-1] `TSwapPool::deposit` is missing 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 sent when market conditions are unfavorable to deposit, even when adding a deadline parameter.

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

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

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

[M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol invariant

Description: The TSwapPool contract assumes that the i_wethToken and i_poolToken are standard ERC-20 tokens, where the amount transferred in or out matches the amount specified in safeTransfer and safeTransferFrom calls. However, it does not account for non-standard ERC-20 tokens, such as:

- Rebase Tokens: Tokens that automatically adjust their total supply or individual balances (e.g., Ampleforth), causing the contract's recorded balance (balanceOf(address(this))) to differ from expected reserves.
- Fee-on-Transfer Tokens: Tokens that deduct a fee during transfers (e.g., PAXG), so the amount received by the contract or user is less than the specified amount.
- ERC-777 Tokens: Tokens with callback hooks (e.g., tokensReceived), which could allow reentrancy or unexpected behavior during transfers.

Impact:

- Invariant Breakage: The AMM's constant product invariant (x * y = k) is violated if actual token amounts differ from expected, leading to incorrect pricing, liquidity calculations, or swap outputs. This could allow arbitrageurs to drain reserves or cause losses for liquidity providers.
- Financial Loss: Users may receive fewer tokens than expected (e.g., in swaps or withdrawals) or deposit more than intended, reducing their returns.
- Reentrancy Risk (ERC-777): If i_wethToken or i_poolToken is an ERC-777 token, its callbacks could reenter the contract, potentially manipulating state or causing unexpected behavior (though SafeERC20 mitigates some risks by using non-reentrant functions).
- Protocol Disruption: Repeated transactions with such tokens could compound errors, making the pool unusable or unfairly skewed.

Proof of Concept: Consider a fee-on-transfer token as i_poolToken with a 1% transfer fee:

In deposit, a user calls with `wethToDeposit = 1000` and `maximumPoolTokensToDeposit = 1000`. The contract calculates `poolTokensToDeposit = 1000` (via getPoolTokensToDepositBasedOnWeth). During `_addLiquidityMintAndTransfer`, `i_poolToken.safeTransferFrom(msg.sender, address(this), 1000)` is called. Due to a 1% fee, only 990 pool tokens are received, but the contract assumes 1000. The invariant `k = wethReserves * poolTokenReserves` is calculated based on 1000 pool tokens, not 990, leading to incorrect liquidity token minting and future pricing errors.

For rebase tokens, if `i_poolToken` rebases down after a deposit, `i_poolToken.balanceOf(address(this))` decreases unexpectedly, breaking the invariant in subsequent swaps. For `ERC-777`, a malicious tokensReceived callback could attempt reentrancy, though `SafeERC20` reduces this risk by avoiding direct call.

Proof of Code

Mock fee-on-transfer token

```
    contract FeeToken is ERC20 {
        constructor() ERC20("FeeToken", "FEE") {
            _mint(msg.sender, 1_000_000);
        }
        function transfer(address recipient, uint256 amount) public
            override returns (bool) {
            uint256 fee = amount / 100; // 1% fee
            _burn(msg.sender, fee);
            _transfer(msg.sender, recipient, amount - fee);
            return true;
        }
        function transferFrom(address sender, address recipient,
            uint256 amount) public override returns (bool) {
            uint256 fee = amount / 100;
            _burn(sender, fee);
            _transfer(sender, recipient, amount - fee);
```

```
                return true;
            }
        }
```

Place this test and mock fee-on-transfer token in `TSwapPool.t.sol`

```
    function testFeeOnTransferBreaksInvariant() public {
        FeeToken feeToken = new FeeToken();

        uint256 wethToDeposit = 1e18;
        uint256 maxPoolTokens = 1e18;

        pool = new TSwapPool(address(feeToken), address(weth), "TokenA"
            , "LA");

        feeToken.mint(user, maxPoolTokens);

        vm.startPrank(user);
        feeToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.deposit(wethToDeposit, 0, maxPoolTokens, uint64(block.
            timestamp));
        vm.stopPrank();

        // Check reserves
        uint256 wethReserves = 1e18; // Mocked
        uint256 feeTokenReserves = feeToken.balanceOf(address(pool));
        assertEq(feeTokenReserves, 0.99e18, "Expected tokens after fee"
            );

        // Invariant check: k = wethReserves * feeTokenReserves
        // Expected k (if no fee): 1e18 * 1e18 = 1e36
        // Actual k: 1e18 * 0.99e18 = 0.99e36
        uint256 actualK = wethReserves * feeTokenReserves;
        assertEq(actualK, 0.99e36, "Invariant broken due to fee-on-
            transfer");
    }
```

Recommended Mitigation:

1. Measure balance changes: update transfer functions to measure `balanceOf(address(this))` before and after transfers to determine actual amounts received or sent.

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

```

+        uint256 wethBalanceBefore = i_wethToken.balanceOf(address(this));
+        uint256 poolTokenBalanceBefore = i_poolToken.balanceOf(address(this));

         i_wethToken.safeTransferFrom(msg.sender, address(this), wethToDeposit);
         i_poolToken.safeTransferFrom(
             msg.sender,
             address(this),
             poolTokensToDeposit
         );

+        uint256 wethReceived = i_wethToken.balanceOf(address(this)) - wethBalanceBefore;
+        uint256 poolTokensReceived = i_poolToken.balanceOf(address(this)) - poolBalanceBefore;
+        require(wethReceived >= wethToDeposit, "WETH transfer failed or fee-on-transfer");
+        require(poolTokensReceived >= poolTokensToDeposit, "Pool token transfer failed or fee-on-transfer");
     }
```

Apply similar logic in `_swap`, `withdraw`, etc.

2. Restrict Token Types: add a check in the constructor or a configuration function to ensure `i_wethToken` and `i_poolToken` are standard ERC-20 tokens (e.g., reject known rebase/ERC-777 tokens via a registry or interface check)
3. Disable `ERC-777` Callbacks: explicitly use `SafeERC20`'s non-callback methods or add reentrancy guards (though `SafeERC20` already mitigates much of this).

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameteres out of order

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

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

Recommended Mitigation:

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

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

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

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return value `output` it is never assigned a value, nor uses an explicit return statement.

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

Proof of Concept:

Place this code in `TSwapPool.t.sol`

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

        vm.startPrank(user);
        uint256 initialUserWethBalance = weth.balanceOf(user);
        poolToken.approve(address(pool), 10e18);
        uint256 userSpentPoolToken = pool.swapExactInput(
            poolToken,
            10e18,
            weth,
            0,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        uint256 userSpentWeth = weth.balanceOf(user) -
           initialUserWethBalance;
        assertNotEq(userSpentWeth, 0);
        assertEq(userSpentPoolToken, 0);
    }
```

Recommended Mitigation:

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

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

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

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

Info

[I-1] Lack of zero address validation

Check for `address(0)` when assigning values to address state variables.

Foundings

- Found in src/PoolFactory.sol Line: 43

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

- Found in src/TSwapPool.sol Line: 99

```
 +      if (wethToken == address(0)) {
```

```
+       revert();
+   }
    i_wethToken = IERC20(wethToken);
```

- Found in src/TSwapPool.sol Line: 100

```
+   if (poolToken == address(0)) {
+       revert();
+   }
    i_poolToken = IERC20(poolToken);
```

[I-2] `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed

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

[I-3] `PoolFactory::createPool` should 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] Event `TSwapPool::Swap` is missing `indexed` fields

Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

Foundings

- Found in src/TSwapPool.sol Line: 52
- Found in src/TSwapPool.sol Line: 57
- Found in src/TSwapPool.sol Line: 63

[I-5] Function `TSwapPool::swapExactInput` should be `external` instead of `public` because it is not used internally

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

[I-6] Missing natspec for the `TSwapPool::swapExactInput` function

Need to add natspec as done for the `TSwapPool::swapExactOutput` function.

[I-7] Missing natspec for the `deadline` parameter in the `TSwapPool::swapExactOutput` function

```
    /*
     * @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
+    * @param deadline The deadline for the transaction to be completed
    by
     */
    function swapExactOutput(
```

[I-8] Using magic numbers instead of constants

Define and use `constant` variables instead of using literals. If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

```
+   uint256 private constant WITHOUT_FEE_MULTIPLIER = 997;
+   uint256 private constant WITH_FEE_MULTIPLIER = 1000;
+   uint256 private constant INPUT_AMOUNT = 1e18;
+   uint256 private constant EXTRA_INCENTIVE = 1
   _000_000_000_000_000_000;
```

Foundings

- Found in src/TSwapPool.sol Line: 287

```
-   uint256 inputAmountMinusFee = inputAmount * 997;
+   uint256 inputAmountMinusFee = inputAmount * WITHOUT_FEE_MULTIPLIER;
```

- Found in src/TSwapPool.sol Line: 289

```
-   uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
+   uint256 denominator = (inputReserves * WITH_FEE_MULTIPLIER) +
   inputAmountMinusFee;
```

- Found in src/TSwapPool.sol Line: 307

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

- Found in src/TSwapPool.sol Line: 428

```
-   outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
+   outputToken.safeTransfer(msg.sender, EXTRA_INCENTIVE);
```

- Found in src/TSwapPool.sol Line: 482

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

- Found in src/TSwapPool.sol Line: 492

```
    function getPriceOfOnePoolTokenInWeth() external view returns (
        uint256) {
        return
```

```
            getOutputAmountBasedOnInput(
-                1e18,
+                INPUT_AMOUNT,
                 i_poolToken.balanceOf(address(this)),
                 i_wethToken.balanceOf(address(this))
            );
    }
```

[I-9] It's better to follow the CEI (Checks, Effects, and Interactions) principle in the else branch of the `TSwap::deposit` function

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

Gas

[G-1] Function `TSwapPool::_isUnknown` can be simplified

```
    function _isUnknown(IERC20 token) private view returns (bool) {
-       if (token != i_wethToken && token != i_poolToken) {
-           return true;
-       }
-       return false;
+       return token != i_wethToken && token != i_poolToken;
    }
```

[G-2] Local variable `poolTokenReserves` in the function `TSwapPool::deposit` is not used

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