TSwap Audit Report

Version 1.0

Jack

July 29, 2025

TSwap Audit Report

Jack

July 29, 2025

Prepared by: Jack Lead Auditors: - Jack

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

TSwap Pools

The protocol starts as simply a `PoolFactory` contract. This contract is used to create new "pools" of tokens. It helps make sure every pool token uses the correct logic. But all the magic is in each `TSwapPool` contract.

You can think of each `TSwapPool` contract as it's own exchange between exactly 2 assets. Any ERC20 and the WETH token. These pools allow users to permissionlessly swap between an ERC20 that has a pool and WETH. Once enough pools are created, users can easily "hop" between supported ERC20s.

For example: 1. User A has 10 USDC 2. They want to use it to buy DAI 3. They `swap` their 10 USDC -> WETH in the USDC/WETH pool 4. Then they `swap` their WETH -> DAI in the DAI/WETH pool

Every pool is a pair of `TOKEN X` & `WETH`.

There are 2 functions users can call to swap tokens in the pool. - `swapExactInput` - `swapExactOutput`

We will talk about what those do in a little.

Disclaimer

The Jack 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

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

- None

Issues found

Severity	Number of issues found
High	3
Medium	2
Low	2
Info	10
Total	17

Findings

High

[H-1] Incorrect Fee Multiplier in `getInputAmountBasedOnOutput` Leading to Excessive Input Charges

Description:
The function

```
function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(outputAmount)
    revertIfZero(outputReserves)
    returns (uint256 inputAmount)
{
    // @audit-high users are charged so much
    // written Literal Instead of Constant
    return
        ((inputReserves * outputAmount) * 10000) /
        ((outputReserves - outputAmount) * 997);
}
```

uses the literal `10000` as a fee multiplier, whereas the protocol's fee model (0.3%) is implemented elsewhere with numerator 997 and denominator 1000. As a result, this function overstates the input required by a factor of ten and ignores the established fee-denominator constant.

Impact:
- Users invoking this quote function will be told to supply up to 10× more input tokens than necessary to obtain the desired `outputAmount`.
- Front-ends and integrators relying on this on-chain quote will display wildly inflated "required input" figures, degrading UX and potentially causing failed transactions (due to insufficient allowance or balance).
- Liquidity consumers lose funds through unnecessary over-spending, undermining trust and possibly blocking arbitrage or other trading strategies.

Proof of Concept:
Assume a pool with
- `inputReserves = 1000 tokens`,

- `outputReserves = 1000 tokens`,
- desired `outputAmount = 100 tokens`.

Correct inverse formula (using feeDenominator = 1000, feeNumerator = 997) yields:

```
numerator   = 1000 * 100 * 1000 = 100_000_000
denominator = (1000 - 100) * 997 = 900 * 997 = 897_300
inputNeeded = 100_000_000 / 897_300 = 111.5 tokens
```

But the flawed code computes:

```
numerator   = 1000 * 100 * 10000 = 1_000_000_000
denominator = 900 * 997 = 897_300
inputNeeded = 1_000_000_000 / 897_300 = 1 114.0 tokens
```

Users are thus quoted ~1 114 tokens instead of the correct ~112 tokens.

Recommended Mitigation:

1. Define and reuse fee constants:

```
solidity uint256 private constant FEE_NUMERATOR = 997; uint256 private constant FEE_DENOMINATOR = 1000;
```

2. Correct the inverse formula to mirror `getOutputAmountBasedOnInput` logic, e.g.:

```
solidity function getInputAmountBasedOnOutput( uint256 outputAmount, uint256 inputReserves, uint256 outputReserves )public pure revertIfZero (outputAmount)revertIfZero(outputReserves)returns (uint256 inputAmount ){ //fee-adjusted numerator/denominator uint256 numerator = inputReserves * outputAmount * FEE_DENOMINATOR; uint256 denominator = (outputReserves - outputAmount)* FEE_NUMERATOR; // round up to ensure enough input return (numerator + denominator - 1)/ denominator; }
```

3. Add NatSpec to clarify that the result is rounded up and includes fees.

4. Add unit tests to verify symmetry between `getOutputAmountBasedOnInput` and its inverse, ensuring no factor-of-ten discrepancies.

[H-2] Unchecked `deadline` Parameter in `deposit` Function

Description:

The `deposit` function signature includes a `uint64 deadline` parameter intended to prevent execution after a certain timestamp, but the function body never checks or enforces it. There is no `revertIfDeadlinePassed(deadline)` modifier or manual comparison against `block.timestamp`, so the deadline is effectively ignored.

```
function deposit(
    uint256 wethToDeposit,
```

```
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    uint64 deadline
)
    external
    revertIfZero(wethToDeposit)
    returns (uint256 liquidityTokensToMint)
{
    //...no check of `deadline`...
}
```

Impact:

- Users relying on `deadline` to guard against adverse price movements or MEV sandwich attacks receive no protection: transactions signed with a soon-to-expire deadline will still execute at any future block.
- If market prices shift unfavorably after the intended deadline, users may deposit at a much worse price or trigger unintended reverts (e.g. hitting `maximumPoolTokensToDeposit`) despite believing their order should have expired.
- Front-ends and integrators will display a false sense of slippage/deadline protection, leading to severe UX disruption and potential financial loss.

Proof of Concept:

1. User signs a transaction:

```
js pool.deposit( parseEther("10"), //wethToDeposit 100, // minimumLiquidityToken
 200, // maximumPoolTokensToDeposit Math.floor(Date.now()/1000)+ 60
// deadline = now + 1 minute );
```

2. The user waits 5 minutes (past their deadline).
3. Market liquidity shifts, pool reserves change.
4. User (or attacker) submits the same transaction on-chain. It still executes because the contract never checks `deadline`.
5. The user ends up depositing under much worse conditions, or unexpectedly reverts on `maximumPoolTokensToDeposit`, despite believing the order had expired.

Recommended Mitigation:

Enforce the `deadline` at the start of `deposit`, for example by adding the existing `revertIfDeadlinePassed` modifier:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
-   uint64 deadline
-) external
+   uint64 deadline
```

```
+) external
+  revertIfDeadlinePassed(deadline)
   revertIfZero(wethToDeposit)
   returns (uint256 liquidityTokensToMint)
{
    // now will revert if block.timestamp > deadline
   ...
}
```

Alternatively, insert an explicit check:

```
if (block.timestamp > deadline) {
    revert TSwapPool__DeadlinePassed(deadline);
}
```

[H-3] Unbounded Incentive Payout on Counter Reset Breaks Pool Invariants

Description:
The private `_swap` function increments `swap_count` on every trade and, once `swap_count` reaches `SWAP_COUNT_MAX`, resets it to zero and blindly transfers a hard-coded `1e18` units of the `outputToken` to the caller as a “bonus.” No budget checks, no per-token limits, no invariant maintenance—just a massive literal payout on every Nth swap:

```
swap_count++;
if (swap_count >= SWAP_COUNT_MAX) {
    swap_count = 0;
    // Huge fixed bonus paid out of pool reserves
    outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
}
```

Impact:
- Every `SWAP_COUNT_MAX`th swap, the pool loses `1e18` tokens of whichever `outputToken` the user requested, potentially draining its reserves.
- An attacker can repeatedly call `_swap` using any whitelisted token as input and drain the corresponding `outputToken` by triggering the bonus.
- The pool’s core constant-product or balance invariants are violated every time the bonus pays out, leading to inaccurate pricing, broken accounting, and loss of user funds.
- Because the literal is oversized and uncontrolled, even a well-capitalized pool can be emptied in a small number of cycles.

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

Proof Of Code

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

```

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

        uint256 outputWeth = 1e17;

        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:

1. Remove or severely limit the "nth-swap bonus." If incentives are required:

- Define a small, configurable bonus constant (e.g. `BONUS_PER_SWAP = 1e15`), not an enormous

hard-coded literal.
- Track a global reward budget and decrement it on each payout so the pool cannot be fully drained.
2. Only issue the bonus after a successful swap and after all invariant checks and balance updates.
3. Emit a dedicated `SwapBonusPaid` event so off-chain tooling can monitor and audit reward emissions.
4. Document the maximum possible payout per swap and enforce it via a `require(bonus <= maxBonusPerSwap)` check.
5. Add unit tests to simulate high-frequency swapping and verify that reserves never fall below the minimum threshold.

Medium

[M-1] Incorrect Function Call in TSwapPool::sellPoolTokens Leading to Wrong Return Value

Description:
The `sellPoolTokens` wrapper is intended to let users sell a specified amount of pool tokens in exchange for WETH, returning the amount of WETH received. Instead, it mistakenly calls `swapExactOutput`, treating the input `poolTokenAmount` as the **desired WETH output** rather than the exact pool-token input:

```
1 function sellPoolTokens(uint256 poolTokenAmount) external returns (
      uint256 wethAmount) {
2     return swapExactOutput(
3         i_poolToken,          // inputToken
4         i_wethToken,          // outputToken
5         poolTokenAmount,      // treated as desiredOutputAmount (wrong)
6         uint64(block.timestamp)
7     );
8 }
```

Impact:
• High likelihood of misbehavior: callers expect to sell `poolTokenAmount` tokens and receive WETH, but the contract instead spends the minimum necessary pool tokens to return exactly `poolTokenAmount` WETH, then returns the **pool tokens spent**.
• User confusion and potential loss: a user calling `sellPoolTokens(100 ether)` may end up spending far more or fewer pool tokens than intended, or receive no WETH at all if the pool lacks sufficient liquidity.
• Documentation mismatch: NatSpec describes selling tokens for WETH, yet the actual behavior is inverted.

Proof of Concept:

```
// User expects to sell 100 pool tokens and get WETH back:
const receivedWeth = await pool.sellPoolTokens(parseEther("100"));
// Internally calls swapExactOutput(..., desiredOutput=100e18)
// -> returns amount of pool tokens spent
// -> `receivedWeth` is actually pool tokens spent, not WETH received
```

Recommended Mitigation:

Replace the call to `swapExactOutput` with `swapExactInput`, passing `poolTokenAmount` as the exact input and specifying an appropriate `minOutputAmount`. Update NatSpec accordingly:

```
function sellPoolTokens(
    uint256 poolTokenAmount
) external returns (uint256 wethAmount) {
-    return swapExactOutput(
-        i_poolToken,
-        i_wethToken,
-        poolTokenAmount,
-        uint64(block.timestamp)
-    );
+    /// @notice Sell exactly `poolTokenAmount` pool tokens for WETH.
+    /// @param poolTokenAmount Amount of pool tokens to sell.
+    /// @return wethAmount      Amount of WETH received.
+    return swapExactInput(
+        i_poolToken,
+        poolTokenAmount,    // exact input amount
+        i_wethToken,
+        0,                  // minOutputAmount: 0 or a stricter minimum
+        uint64(block.timestamp)
+    );
}
```

[M-2] Missing Slippage Protection in `swapExactOutput`

Description

The `swapExactOutput` function lets users specify the exact `outputAmount` they wish to receive but does not allow them to cap the maximum `inputAmount` they are willing to spend. As a result, if the pool's reserves change between the time a user signs the transaction and the time it is mined (e.g. due to front-running or other trades), the required input can increase arbitrarily and the user will have no on-chain safeguard.

```
function swapExactOutput(
    IERC20 inputToken,
    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));

    // Compute required input at current reserves
    inputAmount = getInputAmountBasedOnOutput(
        outputAmount,
        inputReserves,
        outputReserves
    );

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

Impact

- A user who expects to spend roughly *X* input-tokens may end up spending much more if another transaction shifts the price just before theirs is executed.
- Front-runners or MEV bots can manipulate the pool prior to a victim's transaction, forcing them to overpay without any way to revert.
- This undermines UX and exposes traders to unbounded downside risk on "exact output" trades.

Proof of Concept

1. Pool state before any trades:
- 10 000 USDC in reserves
- 5 000 WETH in reserves
2. Victim constructs `swapExactOutput(USDC, WETH, 10 WETH, deadline)` and signs, expecting to pay ~2014 USDC.
3. Attacker sees the pending tx and executes a small USDC→WETH trade just before it, reducing USDC reserve and increasing WETH reserve.
4. When the victim's transaction is mined, `getInputAmountBasedOnOutput` now returns ~2200 USDC. The victim has no way to say "I only want to spend <= 2100 USDC," so they are forced to pay the inflated amount.

Recommended Mitigation

1. Add an explicit `uint256 maxInputAmount` parameter to the function signature:
`solidity function swapExactOutput( IERC20 inputToken, IERC20 outputToken , uint256 outputAmount, uint256 maxInputAmount, uint64 deadline ) public ...returns (uint256 inputAmount){ ...}`
2. After computing `inputAmount`, enforce:

`solidity require(inputAmount <= maxInputAmount, "TSwapPool: Slippage exceeded");`

3. Update frontend and SDKs to collect and pass the user's desired slippage tolerance.
4. Add unit tests to verify that transactions revert whenever `inputAmount` exceeds `maxInputAmount`, preventing unexpected overpayment.

Low

[L-1] Unused Return Value in TSwapPool::swapExactInput

Description:
The function `swapExactInput` declares a return value `uint256 output` but none of its internal or external callers capture or use this value. As a result, callers receive no feedback on the actual amount of output tokens swapped, and any downstream code relying on this return will be operating on a default zero.

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    // @audit-low the return value is not used
    // IMPACT: protocol is giving the wrong return
    // LIKELYHOOD: high
    returns (uint256 output)
{
    //...perform swap, compute `output`...
}
```

Impact:
• Callers (including wrappers like `sellPoolTokens`) receive no actual swap result, leading to misleading or zero return values.
• User interfaces and integrators cannot programmatically verify swap outcomes, increasing risk of UX bugs and reconciliation errors.
• Even if the swap succeeds on-chain, off-chain tooling will report no tokens received.

Recommended Mitigation:
1. Ensure all callers capture and propagate the returned `output` value:

`diff - pool.swapExactInput(...); + uint256 received = pool.swapExactInput (...); + return received;` 2. Update wrapper functions to return or emit the actual `output`. 3. If the return is truly unnecessary, remove it from the signature to avoid confusion.

[L-2] Incorrect Parameter Order in LiquidityAdded Event

Description:
In `_addLiquidityMintAndTransfer`, the `LiquidityAdded` event is emitted with the WETH and pool-token amounts swapped:

```
emit LiquidityAdded(
    msg.sender,
    poolTokensToDeposit,  // should be wethToDeposit
    wethToDeposit         // should be poolTokensToDeposit
);
```

The event is declared as:

```
event LiquidityAdded(
    address indexed liquidityProvider,
    uint256 wethDeposited,
    uint256 poolTokensDeposited
);
```

Impact:
• Off-chain tooling (analytics dashboards, indexers, subgraphs) will record incorrect deposit amounts.
• Users and integrators relying on event data will see mismatched WETH vs. pool-token values.
• Troubleshooting and reconciliation of on-chain vs. off-chain records will be confusing.

Recommended Mitigation:
Swap the two numeric arguments when emitting:

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

Verify off-chain consumers now receive the correct (`wethDeposited, poolTokensDeposited`) ordering.

Informational

[I-1] Error `PoolFactory::PoolFactory__PoolDoesNotExist` not used, should be deleted

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

[I-2] Missing Zero-Address Check in Constructor

Description:
The constructor accepts an `address wethToken` parameter but does not validate it. As a result, if the zero address (`0x0000000000000000000000000000000000000000`) is provided, the contract's internal reference `i_wethToken` will be set to zero. Any subsequent operations assuming a valid WETH contract (e.g., wrapping, unwrapping, ERC-20 transfers) will then fail or revert.

Impact:
• Denial of Service – calls to WETH functions (deposit, withdraw, transfer) will always revert when using the zero address, effectively breaking core functionality.
• Fund Lock – users may send ETH expecting to receive WETH, but the contract will revert and lock their funds.
• Misconfiguration Risk – attacker or deployer mistake can render the contract unusable immediately upon deployment.

Recommended Mitigation:
Add explicit validation in the constructor to ensure `wethToken` is non-zero (and optionally has contract code):

```
constructor(address wethToken) {
    require(wethToken != address(0), "WETH address cannot be zero");
    uint256 size;
    assembly { size := extcodesize(wethToken) }
    require(size > 0, "WETH address has no code");
    i_wethToken = wethToken;
}
```

[I-3] Incorrect Use of `name()` Instead of `symbol()` in Liquidity Token Symbol

Description:
When constructing the `liquidityTokenSymbol`, the code calls `IERC20(tokenAddress).name()` rather than `IERC20(tokenAddress).symbol()`. As a result, the symbol for the newly minted liquidity token is based on the token's full name, not its shorter ticker symbol.

Impact:
- UX Confusion – end users and integrations will see unexpected or overly long symbols (e.g. "tsUSD Coin" instead of "tsUSDC"), leading to confusion and potential mis-tracking of positions.

- Integration Breakage – front-ends, wallets, or analytics tools expecting the conventional ticker symbol may misidentify the pool token or fail to display it at all.
- Branding Inconsistency – liquidity token naming deviates from industry norms, undermining clarity and trust.

Recommended Mitigation:
Replace the erroneous `.name()` call with `.symbol()` when building the symbol string:

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

Optionally, validate both `name()` and `symbol()` return non-empty strings to guard against non-standard tokens.

[I-4] Ambiguous Numeric Literal for `MINIMUM_WETH_LIQUIDITY`

Description:
The constant

```
uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
```

is declared without any indication of its unit—wei, WETH (18-decimal) tokens, or an abstract "amount." Underscores improve readability, but the absence of a comment or a unit suffix makes it unclear whether this value represents 1 Gwei, 1 billion WETH, or something else entirely.

Impact:
- Misconfiguration Risk: If interpreted as wei, the minimum liquidity is merely 0.000000001 ETH—practically zero—undermining any "minimum liquidity" protection.
- Denial of Service: If intended as 1 billion WETH (i.e. 1e9 × 1e18 wei), ordinary deposits (e.g. 1 WETH) will always fall short and revert, breaking pool creation or swaps.
- Economic Exploit: An attacker could spin up pools with negligible WETH backing (if treated as wei) and manipulate price or drain assets via flash swaps.

Recommended Mitigation:
1. Add a clarifying comment next to the declaration to document the intended unit and rationale.

[I-5] Missing Zero-Address Validation in TSwapPool Constructor

Description:
TSwapPool::constructor accepts two raw addresses—`poolToken` and `wethToken`—and immedi-

ately casts them to `IERC20` without checking for the zero address. Consequently, if either parameter is provided as `address(0)`, the immutable variables
- TSwapPool::i_poolToken
- TSwapPool::i_wethToken
will point to a non-contract, causing all downstream calls (ERC-20 transfers, deposits, withdrawals) to revert or misbehave.

Impact:
- Denial of Service – core pool functionality (swaps, liquidity updates) will always revert when interacting with a zero-address token, rendering the pool unusable.
- Fund Lock – users may mistakenly send tokens or WETH into the pool contract expecting proper behavior, only to have transactions revert and funds become inaccessible.
- Misconfiguration Risk – deployment via an external factory or manual deployment with a bad parameter will create a “dead” pool that cannot be repaired.
- Security Exploit – an attacker controlling factory parameters could spin up bogus pools to mislead integrations or front-ends, create phishing vectors, or flood the network with unusable pools.

Recommended Mitigation:
In TSwapPool::constructor, validate both addresses before casting:

```
constructor(
    address poolToken,
    address wethToken,
    string memory liquidityTokenName,
    string memory liquidityTokenSymbol
) ERC20(liquidityTokenName, liquidityTokenSymbol) {
+   require(poolToken != address(0), "TSwapPool: poolToken is zero address");
+   require(wethToken != address(0), "TSwapPool: wethToken is zero address");
+   uint256 size;
+   assembly { size := extcodesize(poolToken) }
+   require(size > 0, "TSwapPool: poolToken has no code");
+   assembly { size := extcodesize(wethToken) }
+   require(size > 0, "TSwapPool: wethToken has no code");

    i_wethToken = IERC20(wethToken);
    i_poolToken = IERC20(poolToken);
}
```

Optionally, include comments documenting the intent and expected token behavior.

[I-6] Emitting Constant in Error Parameters

Description:
In TSwapPool, when checking `wethToDeposit < MINIMUM_WETH_LIQUIDITY`, the code reverts with a custom error that includes the compile-time constant `MINIMUM_WETH_LIQUIDITY` as an argument:

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

Because `MINIMUM_WETH_LIQUIDITY` is immutable by definition, emitting it on every revert is redundant.

Impact:
- Increased gas cost on revert-each extra error argument adds bytes to the revert payload, raising the refund cost for callers.
- Larger on-chain/external data footprint-nodes and indexers must process and store the constant repeatedly, wasting bandwidth and storage.
- Noise in diagnostics-off-chain tooling will see a repeated constant value it already knows, making logs less succinct.

Recommended Mitigation:
Redefine the error to accept only the dynamic value, and omit the constant from the revert. For example:

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

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

Optionally document `MINIMUM_WETH_LIQUIDITY` in NatSpec or a comment so that its value re-

mains discoverable without emitting it at runtime.

[I-7] Superfluous code in `TSwapPool::deposit`

Description: The variable poolTokenReserves is not used in the following code, which is superfluous;

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

Recommended Mitigation: Remove the variable poolTokenReserves.

[I-8] Use of Hard-Coded Numeric Literals Instead of Named Constants

Description:
Throughout TSwapPool.sol, several arithmetic operations and transfers rely on “magic numbers” written directly in the code instead of referring to named constants. Examples include:

- TSwapPool::getOutputAmountBasedOnInput

```
uint256 inputAmountMinusFee = inputAmount * 997;
uint256 denominator          = (inputReserves * 1000) +
    inputAmountMinusFee;
```

- TSwapPool::getInputAmountBasedOnOutput

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

- TSwapPool::swap (on swap_count reset)

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

- TSwapPool::getPriceOfOneWethInPoolTokens and ::getPriceOfOnePoolTokenInWeth

```
getOutputAmountBasedOnInput(
    1e18,
    i_wethToken.balanceOf(address(this)),
    i_poolToken.balanceOf(address(this))
);
```

Embedding these literals directly: 1. Obscures their meaning (fee numerator vs. denominator, base unit).

2. Hampers future updates (e.g. changing fee rates or base-unit).
3. Increases risk of typos and inconsistent values across functions.

Impact:
- Maintainability – new developers or auditors must infer the purpose of each number.
- Consistency – accidental use of the wrong literal (e.g. 1000 vs. 10000) can lead to incorrect pricing or fee calculations.
- Flexibility – hard to adjust protocol parameters (fee, base unit, transfer amount) without error-prone manual replacements.
- Auditability – magic numbers reduce clarity in formal reviews, raising the chance of overlooking a critical parameter.

Recommended Mitigation:
Define descriptive constants at the top of TSwapPool.sol and replace all literals:

```
// Fee parameters for 0.3% swap fee
uint256 private constant FEE_NUMERATOR   = 997;
uint256 private constant FEE_DENOMINATOR = 1000;

// Scale factor used in reserve/output inversion
uint256 private constant PRICE_SCALE     = 10_000;

// Base unit for 1 token (18 decimals)
uint256 private constant BASE_UNIT       = 1 ether;

// Default transfer amount on swap reset
uint256 private constant RESET_TRANSFER_AMOUNT = 1 ether;
```

Then update usages:

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

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

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

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

```
- getOutputAmountBasedOnInput(1e18,..)
+ getOutputAmountBasedOnInput(BASE_UNIT,..)
```

By centralizing these parameters as constants, the code becomes self-documenting, easier to audit, and simpler to adjust.

[I-9] Function Visibility Should Be External for `TSwapPool::totalLiquidityTokenSupply`

Description:
TSwapPool::totalLiquidityTokenSupply is currently declared as a `public` view function but is never called internally. Solidity generates both an external dispatch and an internal function pointer for `public` methods, increasing contract size and gas costs.

Impact:
- Increased deployment and call-site gas: extra dispatcher code for public functions.
- Minor inefficiency on each external invocation of this accessor.

Recommended Mitigation:
Change the visibility from `public` to `external`:

```
-    function totalLiquidityTokenSupply() public view returns (uint256) {
+    function totalLiquidityTokenSupply() external view returns (uint256) {
         return totalSupply();
     }
```

This removes the unnecessary internal invocation stub and reduces both bytecode size and per-call gas.

[I-10] Function Visibility and Missing NatSpec for TSwapPool::swapExactInput

Description:
TSwapPool::swapExactInput is declared as `public` but is never called from within the contract. Public functions incur both an external dispatcher and an internal function pointer, increasing bytecode size and per-call gas. Additionally, the function lacks NatSpec comments, leaving its purpose, parameters, return values, and side-effects undocumented for integrators and auditors.

Impact:
- Gas & Size Overhead: unnecessary internal stub increases deployment and invocation costs.
- Poor UX & Auditability: absence of NatSpec hinders automated documentation generation and makes it harder for developers, front-ends, and auditors to understand required parameters, expected behavior, and error conditions.

Recommended Mitigation:

1. Change visibility to `external` since the function has no internal callers: `diff - function swapExactInput( + /// @notice Swaps an exact amount of input tokens for output tokens. + /// @param inputToken Address of the ERC-20 token being sold. + /// @param inputAmount Exact amount of `inputToken` to swap. + /// @param outputToken Address of the ERC-20 token to receive. + /// @param minOutputAmount Minimum acceptable amount of `outputToken`. + /// @param deadline Unix timestamp after which the swap reverts. + /// @return outputAmount Amount of `outputToken` received. - IERC20 inputToken, - uint256 inputAmount, - IERC20 outputToken, - uint256 minOutputAmount, - uint64 deadline - )public returns (uint256){ + )external returns ( uint256 outputAmount){ // function body...}`