TSwap Audit Report

Version 1.0

Albert Kacirek

November 11, 2025

Contents

Disclaimer

While every effort has been made to identify vulnerabilities within the time allotted, no audit can guarantee the absence of defects. This assessment is not an endorsement of the underlying business or product. The review focused exclusively on the security aspects of the Solidity implementation provided.

Protocol Summary

TSwap is a permissionless, constant-product Automated Market Maker (AMM) that enables users to swap between any ERC-20 token and WETH. Instead of using an order book, each `TSwapPool` contract maintains reserves of the two assets and determines prices using the invariant `x * y = k`, where `x` and `y` represent token balances in the pool.

Users can trade by providing either: - an exact input amount (`swapExactInput`) and receiving the resulting output amount, or - an exact output amount (`swapExactOutput`) and paying whatever input amount the pool requires.

Liquidity Providers (LPs) deposit an equal-value amount of the ERC-20 token and WETH to earn 0.3% of swap volume, proportional to their share of total liquidity. LP shares are represented by ERC-20 LP tokens minted directly by each pool.

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.

Audit Details

The findings in this document correspond to this commit hash:

```
e643a8d4c2c802490976b538dd009b351b1c8dda
```

Scope

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

Executive Summary

The codebase implements a simplified AMM similar to Uniswap V1, but several critical issues in the current implementation allow attackers or misconfigured tokens to break the constant-product invariant and drain liquidity. The most severe problems arise from:

- direct token giveaways during swaps,
- missing slippage protections,
- incorrect exact-output fee math,
- unsafe deposit logic that mints LP tokens prior to receiving assets, and
- lack of handling for fee-on-transfer or non-standard ERC-20 tokens.

If deployed as-is, liquidity providers' funds are at high risk. To be safe for mainnet use, the protocol must address the invariant-breaking behaviors, enforce proper CEI and `nonReentrant` protections, correct the swap fee formulas, and ensure initial liquidity and token transfer accounting are robust.

A second audit and full differential testing are strongly recommended after applying mitigation changes.

Risk Classification

↓Likelihood / Impact →	**High**	**Medium**	**Low**
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

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

Issues found

Severity	Number of issues found
High	7
Medium	2
Low	2
Info	4
Gas	2
Total	17

Findings

High

[H-1] Extra tokens given to users after every `swap_count` breaks protocol invariant `x * y = k` in `TSwapPool::_swap`

Description:

The protocol follows a strict invariant of `x * y = k`, where: - `x` is the balance of pool tokens - `y` is the balance of WETH - `k` is the product of the two balances

The pool follows the constant-product invariant `x * y = k`, meaning swaps must not change the product of reserves except through defined fees. Giving away tokens directly reduces one reserve without corresponding input and therefore lowers `k`.

The following block of code in the `_swap` function is responsible for the issue:

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

Impact:

A user can perform repeated swaps to continuously collect the bonus, draining pool reserves until liquidity is exhausted. This is a direct, deterministic loss of funds for liquidity providers.

Proof of Concept:

1) A user swaps 10 times and grabs the incentive of `1_000_000_000_000_000_000` tokens
2) The user can keep doing this until all protocol funds are drained

To verify this, place the Foundry test below into the `TSwapPool.t.sol` test file:

```
    function testInvariantBreaks() 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); // using a mock/mintable test token
        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(outputWeth + 1e18);

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

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

Recommended Mitigation:

Remove the token incentive mechanism. If you wish to keep this functionality, you should account for the breakage of the `x * y = k` protocol invariant or set aside tokens in a similar way as with fees. If incentives are desired, they must be funded from a separate treasury or from accrued protocol fees rather than directly from liquidity provider reserves.

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

Also ensure all `deposit`, `withdraw`, and `swap` functions follow Checks–Effects–Interactions (CEI) and are wrapped in `nonReentrant` to prevent ERC777-style reentrancy during token transfers.

[H-2] Pool breaks with fee-on-transfer or non-standard ERC-20 tokens, breaking the protocol's invariant

Description:

The pool assumes `transferFrom(msg.sender, address(this), amount)` increases the pool's balance by exactly the `amount` specified. This breaks the AMM invariant during `deposit`, `swap`, and `withdraw` flows (any code path that assumes the nominal transferred amount equals the observed balance delta). Many tokens, such as fee-on-transfer (taxed) tokens, burn tokens, rebasing tokens, or tokens with callbacks (ERC777) are non-standard. When one of these tokens is used as the pool token, the pool's internal math still treats the nominal amount as having been added even though the actual balance delta is smaller (or behaves unpredictably). That breaks the `x * y = k` invariant and makes the pool exploitable.

Impact:

1) The invariant `x * y = k` becomes invalid
2) Attackers can profit via repeated swaps and drain the funds in the pool
3) Liquidity providers can lose their funds
4) Price oracles derived from the pool produce incorrect data
5) Composability and integrations that expect canonical ERC-20 behaviour fail

Proof of Concept:

1. `i_poolToken` is a fee-on-transfer token with 2% burn on transfer.
2. User attempts to deposit 100 pool tokens to match the WETH ratio.
3. Internal logic assumes the full amount arrives:

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

4. However, the actual amount received is `poolTokensToDeposit * 0.98`
5. Pool thinks it has more tokens than it does - price calculation wrong
6. Attacker now performs swaps to extract excess WETH arbitrage profit.
7. Attacker repeats swaps (or deposits then swaps) exploiting the inflated accounting until WETH (or the other side) is drained

Recommended Mitigation:

Use balance-delta accounting to ensure the amount actually received matches the expected amount:

```
uint256 before = token.balanceOf(address(this));
token.safeTransferFrom(msg.sender, address(this), expectedAmount);
uint256 received = token.balanceOf(address(this)) - before;

require(received == expectedAmount, "Unsupported token: non-standard
    transfer behavior");
```

[H-3] `TSwapPool::sellPoolTokens` returns wrong swap variant, causing users to sell and receive the incorrect amount of tokens

Description:

The `sellPoolTokens` function is intended to allow users to sell pool tokens for WETH. Users indicate how many pool tokens they intend to sell in the `poolTokenAmount` parameter. However, the function currently miscalculates the amount to swap, as the `TSwapPool::swapExactOutput` function allows users to specify an amount of output tokens rather than input tokens.

The user intending to sell `X` pool tokens may instead request `X` WETH out (or whatever the function treats as output), producing incorrect behavior and unexpected user losses or failures.

Impact:

Users would swap the wrong amount of tokens. This can result in users receiving unexpectedly low WETH output or spending far more pool tokens than intended, leading to direct fund loss, which severely disrupts intended protocol functionality.

For example, when a user calls `sellPoolTokens(10)`, he will request 10 of the other token instead of selling 10 pool tokens. This can lead to: 1) Unexpected overpayment/underpayment 2) Failed transactions if reserves are insufficient 3) UX and accounting errors for integrators and front-end tools

Proof of Concept:

The Foundry test below can be added to the `TSwapPool.t.sol` file and proves the function's return value is actually the input amount:

```
function
    test_sellPoolTokens_MisinterpretsAmount_ExactOutputInsteadOfExactInput()
     public {
    // Arrange: LP seeds balanced liquidity
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    // Give user enough pool tokens so the unintended exact-output call can
        succeed
    // (With the current math/fee typo, required input > 1e18 for 1e18 WETH
        out)
    poolToken.mint(user, 100e18);

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

    uint256 wethBefore = weth.balanceOf(user);
    uint256 poolBefore = poolToken.balanceOf(user);

    // User intends to SELL exactly 1e18 pool tokens for WETH...
    uint256 intendedPoolTokensToSell = 1e18;

    // ...but the buggy implementation treats the parameter as WETH to
        receive (exact OUTPUT).
    uint256 ret = pool.sellPoolTokens(intendedPoolTokensToSell);

    uint256 wethAfter = weth.balanceOf(user);
    uint256 poolAfter = poolToken.balanceOf(user);

    uint256 wethReceived = wethAfter - wethBefore;
    uint256 poolSpent = poolBefore - poolAfter;

    // Assert: WETH received == param (treated as exact OUTPUT)
    assertEq(
        wethReceived,
        intendedPoolTokensToSell,
        "Bug: sellPoolTokens(param) gives param WETH out instead of selling param pool tokens"
    );

    // Assert: User actually spent MORE than intended pool tokens (due to AMM pricing/fees)
    assertGt(
        poolSpent,
        intendedPoolTokensToSell,
        "Bug: user sold more pool tokens than intended due to wrong swap flavor"
    );

    // Assert: return value equals INPUT amount (pool tokens spent), not WETH received
    assertEq(
        ret,
        poolSpent,
        "Bug: sellPoolTokens returns the inputAmount (pool tokens spent), not WETH received"
    );
    assertTrue(
        ret != wethReceived,
        "Bug: return value should not equal WETH received under the current implementation"
    );

    vm.stopPrank();
}
```

Recommended Mitigation:

Use `TSwapPool::swapExactInput` instead of `swapExactOutput`. This also requires changing the `sellPoolTokens` function to accept a new parameter, such as `minWethToReceive`, to be passed to `swapExactInput`. Additionally, consider adding a deadline check to the function to respect the deadline set by user.

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

[H-4] Pool can be drained when reserves are zero due to missing `inputReserves > 0` check in `TSwapPool::getOutputAmountBasedOnInput`

Description:

The `TSwapPool` allows initial deposits that set only one side of reserves because the initial branch of `TSwapPool::deposit`: - mints LP equal to `wethToDeposit`, - uses `maximumPoolTokensToDeposit` as the actual pool-token amount, - doesn't enforce both assets > 0, - ignores `minimumLiquidityTokensToMint` .

If the pool is seeded with WETH only (or pool tokens only), a subsequent swap uses:

```
getOutputAmountBasedOnInput(inputAmount, inputReserves, outputReserves)
```

which doesn't guard `inputReserves == 0`. With `inputReserves == 0`, the function returns `outputReserves` for any positive `inputAmount`, allowing an attacker to drain the entire opposite reserve in a single trade.

Impact:

When one side starts at zero or becomes zero, the other side can be fully drained.

Proof of Concept:

To verify this issue, paste this Foundry test into the `TSwapPool.t.sol` file:

```
function test_ZeroReserveDrain() public {
    // Seed single-sided with WETH only
    vm.startPrank(lp);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 0); // zero other side
    pool.deposit(100e18, 0, 0, uint64(block.timestamp)); // allowed by
        current code
    vm.stopPrank();

    // Attacker swaps minimal pool tokens in to drain all WETH
    vm.startPrank(attacker);
    poolToken.mint(attacker, 1); // any non-zero
    poolToken.approve(address(pool), type(uint256).max);
    // Since inputReserves(poolToken) == 0, this returns outputReserves (all
        WETH)
    pool.swapExactInput(poolToken, 1, weth, 0, uint64(block.timestamp));
    vm.stopPrank();

    assertEq(weth.balanceOf(address(pool)), 0, "All WETH drained");
}
```

Recommended Mitigation:

Enforce both sides > `0` on the initial deposit and use the Uniswap-style initial mint: `lp = sqrt(weth * pool)` (scaled). In `getOutputAmountBasedOnInput`, revert if `inputReserves == 0` (and if outputReserves == 0). Add explicit custom errors.

[H-5] LP tokens minted before transfers and without nonReentrant guard, causing reentrancy risk & inflation in TSwapPool::deposit

Description:

_addLiquidityMintAndTransfer calls _mint and emits LiquidityAdded before pulling tokens via safeTransferFrom. With ERC777-style tokens or malicious tokens, a callback during safeTransferFrom can reenter the contract holding freshly minted LP tokens and execute TSwapPool ::withdraw, draining live reserves before this deposit's tokens arrive (classic CEI violation). The contract lacks a nonReentrant guard across deposit, withdraw, and swaps.

Impact:

LP supply inflation & reserve theft risk. Full pool drain is feasible with malicious or upgradable tokens.

Proof of Concept:

The vulnerability comes from the order of operations inside deposit and _addLiquidityMintAndTransfer :

1) The pool mints LP tokens to the depositor before the depositor's tokens have actually been transferred into the pool.
2) Then the pool calls safeTransferFrom(...) to pull in the pool tokens and WETH.
3) If the pool token is malicious (or an ERC777-style token), it can run code during safeTransferFrom via a token hook (tokensReceived, onTransfer, etc.).
4) Inside that callback, the attacker now already owns the newly minted LP tokens.
5) The attacker can call pool.withdraw(liquidityTokensToBurn, ...) to redeem real WETH and pool tokens before their deposit has actually been completed.
6) When the callback returns, the original deposit transfer continues — but at that point, the attacker has already drained real reserves.

Reentrancy order simplified:

```
deposit()
    _addLiquidityMintAndTransfer()
        _mint(attacker, LP)        // attacker now holds LP tokens
            i_poolToken.safeTransferFrom(attacker - pool, amount)
                malicious token triggers hook:
                    attacker calls withdraw(LP)
                        pool sends real tokens to attacker
```

Result: attacker withdraws real assets without ever finishing their deposit, draining the pool.

Recommended Mitigation:

Add `nonReentrant` to all state-changing external/public functions (`deposit`, `withdraw`, `swapExactInput`, `swapExactOutput`, and `sellPoolTokens`). Reorder these functions to follow the CEI order (Checks - Effects - Interactions): compute amounts, pull tokens first (with balance-delta), then mint/burn and emit events after transfers. Consider blocking ERC777 hooks by using `nonReentrant` and/or a pull-pattern for claims.

[H-6] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` would lead to a 90.03% fee, breaking intended functionality

Description:

The `TSwapPool::getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens the user should deposit given an amount 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, leading to a 90.03% fee.

The function also lacks two common safety checks for exact-output quoting: 1) `require(outputAmount < outputReserves)` to avoid division by zero, 2) `require(inputReserves > 0)` to avoid non-sensical quotes.

Finally, exact-output paths should round up (+1) to ensure sufficient input after integer division.

Impact:

Protocol would take significantly more fees from users than intended, making users dramatically overpay on exact-output swaps.

Additionally, quotes near pool exhaustion can divide by zero (when `outputReserves - outputAmount == 0`).

Without round-up, the swap can underpay the pool, or subsequent logic must add implicit slippage.

Proof of Concept:

Minimal arithmetic example (identical-decimal tokens, tiny trade): - inputReserves = 1_000e18 - outputReserves = 1_000e18 - outputAmount = 1e18

Expected fee of 0.3%: input = 1e18 * 1_000 / 997 = 1.003009e18

Current code fee calculation: input = 1e18 * 10_000 / 997 = 10.03009e18

Foundry test (drop in TSwapPool.t.sol):

```
function test_InputQuote_ExactOutput_FeeTypo() public {
    uint256 inputReserves = 1_000 ether;
    uint256 outputReserves = 1_000 ether;
    uint256 out = 1 ether;

    // Expected with 0.30% fee
    uint256 expected = (inputReserves * out * 1_000) / ((outputReserves -
       out) * 997) + 1;
    // Current buggy implementation (10_000)
    uint256 actualBug = (inputReserves * out * 10_000) / ((outputReserves -
       out) * 997);

    assertGt(actualBug, expected * 9); // ~10x larger due to 10_000 typo
}
```

Recommended Mitigation:

1) Replace 10_000 with 1_000
2) Turn the magic numbers into constants for clarity & consistency with `TSwapPool::getOutputAmountBasedOr`

3) Add input/output reserve sanity checks.
4) Add +1 rounding for exact-output quotes.

```
+ error TSwapPool__InsufficientInputReserves();
+ error TSwapPool__OutputExceedsReserves();

+ uint256 private constant FEE_NUMERATOR   = 997;
+ uint256 private constant FEE_DENOMINATOR = 1_000;

function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(outputAmount)
    revertIfZero(outputReserves)
    returns (uint256 inputAmount)
{
-   return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
   outputAmount) * 997);
+   if (inputReserves == 0) revert TSwapPool__InsufficientInputReserves();
+   if (outputAmount >= outputReserves) revert
   TSwapPool__OutputExceedsReserves();
+   uint256 num = inputReserves * outputAmount * FEE_DENOMINATOR;
+   uint256 den = (outputReserves - outputAmount) * FEE_NUMERATOR;
+   // Round up for exact-output
+   return (num + den - 1) / den;
}
```

[H-7] `TSwapPool::swapExactOutput` lacks slippage protection, allowing unbounded input cost and MEV exploitation

Description:

`swapExactOutput` does not include any slippage protection. This function is similar to `TSwapPool::swapExactInput`, where the function specifies the`minOutputAmount`. The `swapExactOutput` should specify the `maxInputAmount`.

The root cause is that `swapExactOutput` never checks that `outputAmount < outputReserves`, so when the requested output equals or exceeds available reserves, the function should revert but instead continues execution.

Impact:

If market conditions change before the transaction gets processed, the caller may get a much worse price.

Transaction becomes highly vulnerable to sandwich attacks: 1) A malicious searcher can trade before the user to worsen their price. 2) The user pays an inflated input amount. 3) The attacker reverts the price afterward, extracting profit.

Proof of Concept:

Here is a scenario that demonstrates the issue: 1) User submits:

```
swapExactOutput(inputToken = WETH, outputToken = TOKEN, outputAmount = 10
    TOKEN)
```

2) MEV bot front-runs with a swap that increases the TOKEN/WETH price.
3) Now the pool requires 15 WETH for the same 10 TOKEN.
4) Since there is no max-input bound, the tx executes:
5) User pays 15 WETH instead of 9.
6) Bot back-runs to restore price and capture profit.

If a `maxInputAmount` were enforced, the transaction would instead revert, preventing loss.

Recommended Mitigation: Include the `maxInputAmount` check in the `swapExactOutput` function, so the user only has to spend up to a specific amount:

```
+ error TSwapPool__MaxInputAmountExceeded(uint256 maxInput, uint256 needed);

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

+       if (outputAmount >= outputReserves) {
```

```
+            revert TSwapPool__OutputTooLow(outputReserves - 1, outputAmount);
+        }

        inputAmount = getInputAmountBasedOnOutput(outputAmount,
            inputReserves, outputReserves);

+        if (inputAmount > maxInputAmount) {
+            revert TSwapPool__MaxInputAmountExceeded(maxInputAmount, inputAmount);
+        }

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

Medium

[M-1] `TSwapPool::_addLiquidityMintAndTransfer` initially mints `wethToDeposit` instead of `sqrt(x*y)` and ignores user's min bounds

Description:

The initial branch:

```
_addLiquidityMintAndTransfer(wethToDeposit, maximumPoolTokensToDeposit,
    wethToDeposit);
```

mints LP == `wethToDeposit`, disregarding `maximumPoolTokensToDeposit`'s ratio and the user's `minimumLiquidityTokensToMint`. This lets the first depositor arbitrarily set the price and LP share accounting, and grief subsequent liquidity providers.

Impact:

Skewed pricing, unfair LP share distribution, potential economic loss for later liquidity providers.

Proof of Concept:

If the first liquidity provider deposits `1 WETH` and `1e18 POOL`, they still get `1e18 LP` under current code, gaining outsized claim on future deposits/fees. For example, if the first LP deposits 1 WETH and only 1 wei of the pool token, the current implementation still mints 1e18 LP (equal to wethToDeposit), giving the depositor an outsized share compared to the actual ratio provided.

Recommended Mitigation:

On first deposit, require both amounts to be greater than `0` and mint: - `lp = sqrt(wethToDeposit * poolTokensToDeposit)` (scaled to 18 decimals) - Enforce `liquidityTokensToMint >= minimumLiquidityTokensToMint` also for the initial deposit.

[M-2] `TSwapPool::deposit` ignores `deadline`, enabling execution after user's intended cutoff (also leading to MEV/sandwich risk)

Description: The `TSwapPool::deposit` accepts a `deadline` parameter intended to prevent the transaction from executing after a deadline specified by the user. However, the function does not enforce the deadline. As a result, deposits may be executed even after the intended deadline. This defeats common "time-based slippage" protections, which can become a major problem when a deposit goes through during market conditions where the deposit rate is unfavorable.

Impact: Users may receive significantly worse deposit rates than expected if miners or MEV bots delay or reorder the transaction. The function can execute during unfavorable price conditions without the user's consent, violating user expectations of slippage/time protection.

Proof of Concept:

The parameter is unused and there's no `revertIfDeadlinePassed(deadline)` in `TSwapPool::deposit`. The call succeeds even if `deadline < block.timestamp`. A minimal test to verify this:

```
function test_Deposit_IgnoresDeadline() public {
    // arrange: seed pool so totalSupply > 0 (or test initial branch)
    uint64 past = uint64(block.timestamp - 1);
    uint256 weth = 1e18;
    // act: approve tokens, then call deposit with a past deadline
    uint256 minted = pool.deposit(
        weth,
        0,                  // min LP (ignored on initial deposit anyway)
        type(uint256).max,
        past                // <-- should revert, but doesn't
    );
    // assert: minted > 0 means deadline was ignored
    assertGt(minted, 0);
}
```

Solidity compiler also warns about the parameter being unused with this message:

```
Unused function parameter. Remove or comment out the variable name to
    silence this warning.
```

Recommended Mitigation: Add a revert in case the deadline has passed to the `deposit` function:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    // @udit deadline is not being used anywhere, so the deposit function
        lacks intended functionality
    uint64 deadline
)
    external
+   revertIfDeadlinePassed(deadline)
    revertIfZero(wethToDeposit)
    returns (uint256 liquidityTokensToMint)
```

Low

[L-1] `TSwapPool::swapExactInput` always returns `0`, as the return variable `output` is never assigned

Description: The `swapExactInput` function is intended to return the amount of `outputToken` sent by the caller. However, the return variable `output` is never assigned a value.

Impact:

The return value will always be `0`, giving incorrect information to caller. This could also lead to logic errors in composed contracts and incorrect off-chain accounting.

Proof of Concept:

Below is a simple Foundry test to verify that the output is always `0`:

```
function test_swapExactInput_ReturnsZero() public {
    // Arrange: LP seeds liquidity
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    // Act: user swaps exact input; function should (buggily) return 0
    vm.startPrank(user);
    poolToken.approve(address(pool), 10e18);
    uint256 ret = pool.swapExactInput(
        poolToken, // inputToken
        10e18, // inputAmount
        weth, // outputToken
        0, // minOutputAmount
        uint64(block.timestamp)
    );
    vm.stopPrank();

    // Assert: return value is always 0 because `output` is never assigned
    assertEq(ret, 0, "swapExactInput currently returns 0 due to missing
       assignment to `output`");
}
```

Recommended Mitigation:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    // output is not assigned anywhere, so the return value will always be 0
    returns (uint256 output)
{
    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) {
+   if (output < minOutputAmount) {

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

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

[L-2] `TSwapPool::LiquidityAdded` event emits parameters in incorrect order

Description:

When the `TSwapPool::_addLiquidityMintAndTransfer` function emits the `LiquidityAdded` event, the emit logs values in an incorrect order. The order or `poolTokensToDeposit` and `wethToDeposit` should be reversed.

Impact:

Event emission is incorrect, potentially leading to confusion and incorrect parsing by off-chain tools.

Proof of Concept:

Event signature:

```
LiquidityAdded(liquidityProvider, wethDeposited, poolTokensDeposited)
```

Actual emission:

```
LiquidityAdded(liquidityProvider, poolTokensDeposited, wethDeposited)
```

Programs listening to logs would parse:

```
wethDeposited   = poolTokensDeposited
poolTokensDeposited = wethDeposited
```

Example misinterpretation in TheGraph or ethers.js:

```
event.args.wethDeposited // returns pool tokens, not WETH
```

Recommended Mitigation:

Change the order of parameters in the event emit:

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

Informational

[I-1] String parameter `liquidityTokenSymbol` in `PoolFactory::createPool` uses `.name()` instead of `.symbol()`, producing incorrect LP token symbols

Description:

In `PoolFactory::createPool`, the `liquidityTokenSymbol` is set using a token's name instead of its symbol:

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

Token names are often much longer and descriptive, while symbols are intentionally short. Using .name() for a symbol can create LP token symbols that are overly long or poorly formatted.

Impact:

Using `.name()` instead of `.symbol()` may lead to symbols that are too long, potentially creating problems in some user interfaces. This can also reduce readability and break token display alignment in some frontends.

Proof of Concept:

As an example, if `IERC20(tokenAddress).name()` holds the value `Tether USD`, the output of `liquidityTokenSymbol` would be `tsTether USD`. Using `.symbol()` would return `USDT`, so the output would be `tsUSDT`.

Recommended Mitigation:

Replace `.name()` with `.symbol()`:

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

Since some ERC20 tokens do not implement `.symbol()` reliably, consider wrapping it in a try/catch with a fallback:

```
string memory underlyingSymbol;
try IERC20(tokenAddress).symbol() returns (string memory s) {
    underlyingSymbol = s;
} catch {
    underlyingSymbol = "TOKEN";
}
string memory liquidityTokenSymbol = string.concat("ts", underlyingSymbol);
```

[I-2] Missing zero address checks and other sanity checks on `wethToken` allows deployment with invalid addresses

Description:

The `PoolFactory` constructor accepts a `wethToken` address but does not perform any validation on it. This allows deployment with: 1) The zero address 2) A non-contract address (EOA) 3) Or a contract that does not conform to the `IERC20` interface

Entering an invalid token could break all pool interactions.

Impact:

If the contract is deployed with an incorrect `wethToken` address (such as the zero address or a non-ERC20 contract), all pools created by the factory will be misconfigured and may become unusable. This would require redeployment to correct, as there is no upgrade path. The impact is therefore limited to deployment misconfiguration rather than user-side risk.

Proof of Concept:

```
// Deploy factory with zero address
PoolFactory factory = new PoolFactory(address(0));

// Calls createPool will produce pools using address(0)
// which cannot be interacted with safely:
// pool.deposit(...) will revert or transfer tokens to address(0)
factory.createPool(DAI);
```

Recommended Mitigation:

Add validation checks in the constructor:

```
constructor(address wethToken) {
+    if (wethToken == address(0)) revert PoolFactory__InvalidWethToken();
+    if (wethToken.code.length == 0) revert PoolFactory__NotAContract(
    wethToken);
     i_wethToken = wethToken;
}
```

After this, define the missing errors:

```
error PoolFactory__InvalidWethToken();
error PoolFactory__NotAContract(address addr);
```

[I-3] Missing event indexing hinders off-chain filtering and integration

Description:

Both `PoolFactory` and `TSwapPool` define events that do not index key address parameters used for filtering. This saves a bit of gas, but makes it harder for off-chain tools to interact with the contract.

`PoolFactory`

```
// Currently:
event PoolCreated(address tokenAddress, address poolAddress);
// Neither address is indexed - consumers must scan all logs and post-filter
    .
```

`TSwapPool`

```
// Currently
event LiquidityAdded(address indexed liquidityProvider, uint256
    wethDeposited, uint256 poolTokensDeposited);
event LiquidityRemoved(address indexed liquidityProvider, uint256
    wethWithdrawn, uint256 poolTokensWithdrawn);
event Swap(address indexed swapper, IERC20 tokenIn, uint256 amountTokenIn,
    IERC20 tokenOut, uint256 amountTokenOut);
// Only swapper is indexed; tokenIn/tokenOut are not, which impairs "all
    swaps for TOKEN_X" queries.
```

Impact:

The lack of indexing doesn't possess any security risk, but it makes it harder to filter events by token or pool across many deployments. Off-chain indexers must also scan more data, increasing indexing time and cost.

Without indexing `tokenAddress` or `poolAddress`, users of the protocol can't filter via topics and must fetch all `PoolCreated` logs and then post-filter in code. Similarly, to find all swaps involving specific tokens, consumers must fetch every `Swap` and filter by `tokenIn`/`tokenOut` values, which scales poorly as activity grows.

However, each additional indexed topic adds a small per-event gas cost. Indexing addresses is typically worth it, especially for `PoolCreated` and `Swap` events because of the analytics/UX value.

Recommended Mitigation:

For `PoolFactory`, consider indexing both the `tokenAddress` and the `poolAddress`:

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

For `TSwapPool`, consider indexing `tokenIn` and `tokenOut` in the Swap event:

```
- 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-4] Price helpers assume 18 decimals and include fee, which can lead to misleading quotes

Description:

`TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` use `1e18` regardless of underlying decimals and call the fee-inclusive output formula. Displayed price therefore depends on token decimals and includes swap fee effects.

Impact:

Confusing/misleading price displays for non-18-dec tokens; fee-skewed indicative prices.

Proof of Concept:

If pool token has for example 6 decimals, quoting `1e18` units is 1e12 tokens instead of 1 token.

Recommended Mitigation:

Normalize by `decimals()` of each token (with safe fallbacks).

Gas

[G-1] Error `PoolFactory::PoolFactory__PoolDoesNotExist` is not used anywhere and should be removed

Description:

The custom error `PoolFactory::PoolFactory__PoolDoesNotExist` is not used anywhere and can safely be removed to save gas and reduce clutter in the `PoolFactory` contract.

Impact:

This error may slightly increase deployment gas cost. It also clutters the ABI and makes the contract code slightly less readable.

Recommended Mitigation:

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

[G-2] `MINIMUM_WETH_LIQUIDITY` is a constant, so it can safely be removed from `TSwapPool__WethDepositAmountTooLow` parameters

Description:

The `TSwapPool::deposit` function can revert with `revert TSwapPool__WethDepositAmountTooLow (MINIMUM_WETH_LIQUIDITY, wethToDeposit)`. Including `MINIMUM_WETH_LIQUIDITY` as a revert parameter is not necessary because `MINIMUM_WETH_LIQUIDITY` is a constant, meaning anyone can see its value by reading the code. Consider removing it from the revert parameters to save gas.

Impact:

The revert message will cost more gas to display each time the `TSwapPool__WethDepositAmountTooLow` revert gets executed.

Recommended Mitigation:

1) Change the error signature:

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

2) Remove the `MINIMUM_WETH_LIQUIDITY` from the revert message:

```
-   revert TSwapPool__WethDepositAmountTooLow(MINIMUM_WETH_LIQUIDITY,
    wethToDeposit);
+   revert TSwapPool__WethDepositAmountTooLow(wethToDeposit);
```

Conclusion

The reviewed implementation of TSwap is not currently safe for production deployment. Multiple high-severity issues allow reserve imbalance, incorrect pricing, and in several cases deterministic draining of pool liquidity. These issues stem primarily from invariant violations, unsafe token transfer assumptions, and missing slippage and deadline checks.

Before deployment, the following changes are required: - Remove or redesign the bonus token transfer mechanic to avoid altering reserves. - Rework liquidity deposit logic to enforce two-asset initialization and mint LP tokens only after assets are received. - Add `nonReentrant` protections and follow Checks-Effects-Interactions ordering. - Use balance-delta accounting to defend against fee-on-transfer / non-standard ERC-20 tokens. - Correct the fee constants in exact-output swaps and require `maxInputAmount`. - Enforce zero-reserve guards and deadline checks.

Once these issues are resolved, a follow-up audit should verify that the constant-product invariant is preserved across deposits, withdrawals, and swaps.