TSwap Protocol Audit Report

Version 1.0

Cyfrin.io

September 18, 2025

TSwap Protocol Audit Report

Vinay

September 18, 2025

Prepared by: Vinay Lead Auditors: - Vinay

Table of Contents

Protocol Summary

TSwap is a decentralized automated market maker (AMM) protocol that facilitates token swaps and liquidity provision between WETH and an ERC20-based pool token. It allows users to:

- **Provide Liquidity:** Users can deposit WETH and pool tokens into the pool to earn trading fees and receive liquidity tokens representing their share of the pool.
- **Swap Tokens:** Users can swap between WETH and the pool token using two core functions:
  - `swapExactInput` — User specifies the exact amount of input tokens to spend.
  - `swapExactOutput` — User specifies the exact amount of output tokens to receive.
- **Mint / Redeem Pool Tokens:** The protocol mints pool tokens to represent proportional ownership of the liquidity pool and allows redeeming them for underlying assets.
- **Trade Pool Tokens:** Users can buy or sell pool tokens directly through swap functions, enabling dynamic pricing based on the pool's reserves using the constant product formula.

The protocol is designed to behave similarly to Uniswap V2, using the constant product market maker (x * y = k) model, where the price of each asset adjusts algorithmically based on supply and demand within the pool.

Disclaimer

The Vinay's 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

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

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Gas	2
Info	6
Total	15

Findings

High

[H-1] Incorrect Fee Calculation Causes Excessive Input Requirement in `TSwapPool::getInputAmountBasedOnOutput`

Description: The `getInputAmountBasedOnOutput` function miscalculates the amount of input tokens needed to receive a desired output amount. It uses hardcoded magic numbers 997 and 10000 incorrectly in the formula:

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

The logic intends to apply a 0.3% fee but uses 10000 as the scaling factor in the numerator while keeping 997 as the denominator multiplier. This results in a misaligned fee calculation and massively inflates the input amount required.

Impact: - Users are overcharged when performing swaps, effectively paying ~91.3% extra instead of a 0.3% fee. - Since this function likely underpins `swapExactOutput`, this affects core swap functionality and poses high financial risk to all users.

Proof of Concept: As a result, users swapping tokens via the `swapExactOutput` function will pay far more tokens than expected for their trades. This becomes particularly risky for users that provide infinite allowance to the `TSwapPool` contract. Moreover, note that the issue is worsened by the fact that the `swapExactOutput` function does not allow users to sepecify a maximum of input tokens, as is described in another issue in this report.

It's worth nothing that the tokens paid by users are not lost, but rather can be swiftly taken by liquidity providers. Therefore, this contract could be used to trick users, have them swap their funds at unfavourable rates and finally rug pull all liquidity from the pool.

To test this, include the following code in the `TSwapPool.t.sol` file:

```
 function testFlawedSwapExactOutput() public {
        uint256 initialLiquidity = 100e18;
        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();

        // User has 11 pool tokens
        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 11e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);
        vm.startPrank(someUser);

        // Users buys 1 WETH from the pool, paying with pool tokens
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(
            poolToken,
            weth,
            1 ether,
            uint64(block.timeStamp)
        );

        // Initial liquidity was 1:1, so user should have paid ~1 pool token
        // However, it spent much more than that. The user started with 11 tokens, and now only has less than
        assertLt(poolToken.balanceOf(someUser), 1 ether);
        vm.stopPrank();

        // The liquidity provider can rug all funds from the pool now,
        // including those deposited by user.
        vm.startPrank(liquidityProvider);
        pool.withdraw(
            pool.balanceOf(liquidityProvider),
            1, // MinimumWethToWihdraw
            1, // MinimumPoolTokenToWihdraw
            uint64(block.timestamp)
        );

        assertEq(weth.balanceOf(address(pool)), 0);
        assertEq(poolToken.balanceOf(address(pool)), 0);
    }
```

Recommended Mitigation: Use consistent scaling base (1000) and apply the fee properly:

```
 return
-           ((inputReserves * outputAmount) * 10000) /
+           ((inputReserves * outputAmount) * 1000) /
            ((outputReserves - outputAmount) * 997);
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

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

Impact: If market conditions changes before the transaction process, the user would get a much worse swap.

Proof of Concept: 1. The price of 1 WETH right now is 1,000 USDC 2. User inputs a `swapExactOutput` looking for 1 WETH 1. inputToken = USDC 2. outputToken = WETH 3. outputAmount = 1 4. deadline = whatever 3. The function does not offer 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 sentthe protocol 10,000 USDC instead of the expected 1,000 USDC

Recommended Mitigation: We should include a `maxInputAmount` so the user has to spend up to a specific amount, and can predict how much they will spend on the protocol.

```
function swapExactOutput(
        IERC20 inputToken,
+       uint256 maxInputAmount,
.
.
.
         inputAmount = getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );
+       if(inputAmount > maxInputAmount){
+           revert();
+       }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
```

[H-3] `TSwapPool::sellPoolTokens` 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 pool tokens and receive WETH in exchange. Users indicate how many pool 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: To demonstrate the bug, I wrote a unit test that simulates a user attempting to sell `pool tokens` for `WETH`.

1. **Setup liquidity**
   The test begins by providing initial liquidity to the pool using both `WETH` and `poolToken`.
2. **Mint pool tokens to user**
   A test address (`someUser`) is given 500 `poolToken` as their balance.
3. **User attempts to sell 2 pool tokens**
   The user calls `sellPoolTokens(2e18)`, intending to **input** 2 `pool tokens` and receive the correct amount of `WETH`.
4. **Bug triggers**
   Inside `sellPoolTokens`, the implementation incorrectly calls `swapExactOutput`, which interprets the user's `sellingAmount` as the **desired output amount** of `WETH`, instead of the **input amount** of pool tokens.
   As a result, the function pulls enough pool tokens from the pool to give the user **2 full WETH**, instead of selling just 2 pool tokens.
5. **Assertion**
   The test ends by asserting the user's `WETH` balance equals `sellingAmount` (2e18).
   This confirms the bug: the user receives 2 WETH instead of the amount they should get for selling 2 pool tokens.

To test this, include the following code in the `TSwapPool.t.sol` file:

```
    function testBugInSellPoolToken() public {
        uint256 initialLiquidity = 100e18;

        // --- Setup: Provide initial liquidity to the pool ---
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        pool.deposit({
            wethToDeposit: 10 ether,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // --- Setup: Give a user (someUser) 500 pool tokens ---
        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 500e18;
        uint256 sellingAmount = 2e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);

        // --- User tries to sell 2 pool tokens for WETH ---
        vm.startPrank(someUser);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);

        // BUG: This internally calls swapExactOutput (wrong), should
            be swapExactInput
        // User specifies the input amount (2 pool tokens) but function
             treats it as output amount
        // This results in incorrect WETH received
        pool.sellPoolTokens(sellingAmount);
        vm.stopPrank();

        // --- Observe balances after selling ---
        // EXPECTED: User loses 2 pool tokens and receives correct
            amount of WETH
        // ACTUAL: Because of swap mismatch, user receives 2 WETH token
             amount instead of selling 2 pool tokens
        console.log(poolToken.balanceOf(someUser));
        console.log(weth.balanceOf(someUser));
        assertEq(weth.balanceOf(someUser), sellingAmount);
    }
```

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 `swapExactInput`)

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

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

[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 two balances

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.

The following block of code is responsible for the issue.

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

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.

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 protocols 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)));
            // console.log(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));
            // console.log(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 this fees.

```
-    swap_count++;
-    // Fee-on-transfer
-    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, operationrs 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, // LP tokens -> if empty,
            we can pick 100% (100% == 17 tokens)
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
```

Low

[L-1] Incorrect parameter order in `LiquidityAdded` event emission

Description: In `TSwapPool::_addLiquidityMintAndTransfer`, the `LiquidityAdded` event is emitted with swapped parameter order:

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

However, the event is expected to log `(msg.sender, wethToDeposit, poolTokensToDeposit)`.

Impact: This does not affect protocol logic, but it misleads off-chain consumers, analytics tools, or UIs relying on event logs. It provides incorrect information about what amounts were deposited.

Recommended Mitigation: Fix the parameter order in the event emission:

```
- 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.

Recommended Mitigation:

```
      {
          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);
          }

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

Informationals

[I-1] Unused custom error `PoolFactory__PoolDoesNotExist`

Description:
The custom error `PoolFactory__PoolDoesNotExist(address tokenAddress)` is declared in the `PoolFactory` contract but is never used anywhere in the codebase. This increases code clutter and may confuse future maintainers.

Recommended Mitigation:
Remove the unused error from the `PoolFactory` contract to keep the codebase clean and maintainable.

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

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

Description:
The `PoolFactory` constructor sets the `i_wethToken` immutable variable without validating the provided `wethToken` address.
If a zero address is accidentally passed, the contract will deploy successfully but fail at runtime when interacting with WETH.

Impact:
While unlikely in production deployments, passing an invalid address could break core functionality of the factory and all created pools.

Recommended Mitigation:
Add a `require(wethToken != address(0))` check to ensure the constructor cannot be initialized with the zero address.

```
constructor(address wethToken) {
-    i_wethToken = wethToken;
+    require(wethToken != address(0), "Invalid WETH address");
+    i_wethToken = wethToken;
}
```

[I-3] Incorrect use of `name()` instead of `symbol()` when creating liquidity token symbol in `createPool`

Description:
In the `createPool` function, the liquidity token **symbol** is constructed using `IERC20(tokenAddress).name()` instead of `symbol()`.
This may lead to confusing or misleading symbols (e.g., `"tsUSD Coin"` instead of `"tsUSDC"`).

Impact:
This does not affect core logic but negatively impacts user experience and UI integration, since token symbols are expected to be short identifiers.

Recommended Mitigation:
Use the `symbol()` function when constructing `liquidityTokenSymbol`:

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

[I-4] Missing zero-address validation in `TSwapPool` constructor

Description:
The `TSwapPool` constructor sets the `i_wethToken` and `i_poolToken` immutable variables without validating the provided `wethToken`, `poolToken` addresses.
If a zero address is passed, the contract will deploy successfully but will revert on any call that tries to interact with those tokens.

Impact:
Passing an invalid (zero) address could render the pool unusable, breaking swaps, liquidity operations, and potentially locking user funds.

Recommended Mitigation:
Validate that neither wethToken nor poolToken is the zero address before assigning them.

```
constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
-         i_wethToken = IERC20(wethToken);
-         i_poolToken = IERC20(poolToken);
+         require(wethToken != address(0) && poolToken != address(0), "
    Invalid address");
+         i_wethToken = IERC20(wethToken);
+         i_poolToken = IERC20(poolToken);
}
```

[I-5] `liquidityTokensToMint` assignment should be done before external call in `TSwapPool::deposit` to follow CEI

Description: In the `TSwapPool::deposit` function's else branch, the assignment `liquidityTokensToMint = wethToDeposit;` occurs after the external call `_addLiquidityMintAndTransfer`. This breaks the Checks-Effects-Interactions (CEI) pattern.

Impact: While it doesn't cause a functional bug now, following CEI reduces the risk of reentrancy or unexpected state inconsistencies if `_addLiquidityMintAndTransfer` logic changes in the future.

Recommended Mitigation: Move the assignment before the external call.

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

[I-6] Use of magic numbers (997 and 1000) in `TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool::getInputAmountBasedOnOutput`

Description: The function hardcodes 997 and 1000 when applying the 0.3% fee logic:

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

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

Using unexplained magic numbers reduces readability and makes future maintenance error-prone.

Impact: - Reduces code readability and maintainability. - Makes it harder to update the fee in the future.

Recommended Mitigation: Define clearly named constants to replace these numbers, and verify that fee math is consistent between both functions:

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

- uint256 inputAmountMinusFee = inputAmount * 997;
- uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
+ uint256 inputAmountMinusFee = inputAmount * FEE_NUMERATOR;
+ uint256 denominator = (inputReserves * FEE_DENOMINATOR) +
    inputAmountMinusFee;

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

Gas

[G-1] Unnecessary emission of constant value in revert

Description: In the following code, `MINIMUM_WETH_LIQUIDITY` is a constant and does not change:

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

Impact: Including a constant in the revert data increases gas usage unnecessarily without providing additional runtime information.

Recommended Mitigation: Remove the constant from the revert arguments and hardcode it in the error message or documentation instead.

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

Removing it slightly lowers deployment bytecode size and runtime revert cost.

[G-2] Unused `poolTokenReserves` variable increases gas usage

Description: In the `TSwapPool::deposit` function, the line

```
uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

Impact: Slight increase in gas cost for every deposit call.

Recommended Mitigation:

```
- uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```