TSwap Audit Report

Version 1.0

Dewaxindo

November 12, 2025

TSwap Audit Report

Dewangga Praxindo

Nov 12, 2025

Prepared by: Dewaxindo Lead Auditors:

- Dewangga Praxindo

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.

Disclaimer

The DEWAXINDO 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

	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Table 1: Severity matrix by likelihood and impact

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

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Gas	0
Info	4
Total	12

Findings

High

[H-1] Absence of slippage safeguards in `TSwapPool::swapExactOutput` may result in users obtaining substantially fewer tokens than anticipated

Description: The `swapExactOutput` function lacks any form of slippage protection mechanism. Mirroring the approach in `TSwapPool::swapExactInput` where a `minOutputAmount` is required, the `swapExactOutput` function ought to require a `maxInputAmount` parameter.

Impact: Should market conditions shift prior to transaction execution, users may experience significantly inferior exchange rates.

Proof of Concept:

1. At the current moment, 1 WETH trades at 1,000 USDC

2. A user invokes `swapExactOutput` requesting 1 WETH
    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline = any value
3. The function lacks a `maxInputAmount` parameter
4. While the transaction awaits processing in the mempool, substantial market movement occurs. The price shifts dramatically: 1 WETH now costs 10,000 USDC, representing a 10x increase from the user's expectation
5. Upon transaction completion, the user has transferred 10,000 USDC to the protocol rather than the anticipated 1,000 USDC

Recommended Mitigation: Add a `maxInputAmount` parameter enabling users to define their maximum acceptable spending limit, providing them with cost predictability and control.

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

[H-2] Wrong fee computation in `TSwapPool::getInputAmountBasedOnOutput` leads to protocol extracting excessive tokens from users, causing fee overcharges

Description: The `getInputAmountBasedOnOutput` function is designed to determine the required token deposit amount based on a desired output token quantity. Nevertheless, the function contains a calculation error. During fee computation, the amount is multiplied by `10_000` when it should be multiplied by `1_000`.

Impact: The protocol charges users higher fees than intended.

Proof of Concept:

Proof Of Code

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

```
/**
     * @notice Proof of Concept for H-2: Incorrect fee calculation in
↪  getInputAmountBasedOnOutput
     * @dev This test demonstrates that the function multiplies by 10_000
↪  instead of 1_000,
     *      causing users to be charged 10x more than intended
     */
    function testIncorrectFeeCalculationInGetInputAmountBasedOnOutput()
    ↪  public {
        // Setup: Add liquidity to the pool
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        // Test parameters
        uint256 outputAmount = 1e18; // User wants 1 WETH
        uint256 inputReserves = poolToken.balanceOf(address(pool)); //
↪  100e18
        uint256 outputReserves = weth.balanceOf(address(pool)); // 100e18

        // Calculate what the input amount SHOULD be (correct formula with
        ↪  1_000)
        uint256 correctInputAmount = ((inputReserves * outputAmount) *
↪  1_000) /
            ((outputReserves - outputAmount) * 997);

        // Get what the function ACTUALLY returns (incorrect formula with
        ↪  10_000)
        uint256 actualInputAmount = pool.getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

        // Log the difference to clearly show the bug
        console.log("Correct input amount (should be):",
        ↪  correctInputAmount);
        console.log("Actual input amount (bug returns):",
        ↪  actualInputAmount);
```

```
        console.log("Overcharge multiplier:", actualInputAmount * 1e18 /
        ↪  correctInputAmount);

        // The actual amount should be 10x higher than the correct amount
        // because 10_000 / 1_000 = 10
        assertEq(actualInputAmount, correctInputAmount * 10, "Protocol
        ↪  charges 10x more than intended");

        // impact: Perform a swap to show user is charged incorrectly
        vm.startPrank(user);
        poolToken.mint(user, actualInputAmount + 1e18); // Mint enough
↪   tokens
        poolToken.approve(address(pool), actualInputAmount);

        uint256 userPoolTokenBalanceBefore = poolToken.balanceOf(user);
        uint256 userWethBalanceBefore = weth.balanceOf(user);

        // This swap will use the incorrect calculation, charging 10x more
        pool.swapExactOutput(poolToken, weth, outputAmount,
↪   uint64(block.timestamp));

        uint256 userPoolTokenBalanceAfter = poolToken.balanceOf(user);
        uint256 userWethBalanceAfter = weth.balanceOf(user);

        uint256 poolTokensSpent = userPoolTokenBalanceBefore -
↪   userPoolTokenBalanceAfter;
        uint256 wethReceived = userWethBalanceAfter -
↪   userWethBalanceBefore;

        // Verify user received the expected output
        assertEq(wethReceived, outputAmount, "User should receive exact
        ↪  output amount");

        // Verify user spent 10x more than they should have
        assertEq(poolTokensSpent, actualInputAmount, "User spent the
        ↪  incorrect amount");
        assertGt(poolTokensSpent, correctInputAmount, "User spent more than
        ↪  the correct amount");

        // The ratio should be approximately 10x (allowing for rounding)
        // This demonstrates the bug: user pays 10x more than they should
        assertApproxEqRel(
            poolTokensSpent,
```

```
            correctInputAmount * 10,
            1e15, // 0.1% tolerance for rounding
            "BUG CONFIRMED: User was charged approximately 10x the correct
            ↪  amount"
        );
        vm.stopPrank();
    }
```

Recommended Mitigation:

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

[H-3] `TSwapPool::sellPoolTokens` incorrectly pairs input and output tokens, leading to users obtaining wrong token quantities

Description: The `sellPoolTokens` function is designed to facilitate the sale of pool tokens in return for WETH. The `poolTokenAmount` parameter represents the quantity of pool tokens the user wishes to sell. Unfortunately, the function computes the exchange amount incorrectly.

The root cause is that `swapExactOutput` is invoked when `swapExactInput` should be used. Since users provide the precise quantity of input tokens (pool tokens) rather than the output quantity, the wrong swap function is being utilized.

Impact: Users will exchange incorrect token amounts, representing a critical failure of core protocol operations.

Proof of Concept:

Proof Of Code

Place the following into TSwapPool.t.sol:

```
  /**
     * @notice Proof of Concept for H-3: sellPoolTokens incorrectly uses
↪  swapExactOutput instead of swapExactInput
     * @dev sellPoolTokens takes poolTokenAmount as input but calls
↪  swapExactOutput which expects output amount.
     *      This causes incorrect token exchange - it tries to get
↪  poolTokenAmount WETH instead of selling poolTokenAmount tokens.
     */
    function testSellPoolTokensUsesWrongSwapFunction() public {
        // Setup pool with liquidity
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 poolTokensToSell = 10e18;

        // Calculate expected WETH using correct function (swapExactInput)
        uint256 expectedWeth = pool.getOutputAmountBasedOnInput(
            poolTokensToSell,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );

        // Test incorrect behavior: sellPoolTokens uses swapExactOutput
        // The bug causes it to try getting poolTokensToSell WETH, which
         ↪  needs many more pool tokens
        vm.startPrank(user);
        poolToken.mint(user, 200e18); // Mint enough to cover the incorrect
↪  calculation
        poolToken.approve(address(pool), type(uint256).max);

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

        // BUG: sellPoolTokens returns the input amount (pool tokens
         ↪  needed), not WETH received
        // This is because it calls swapExactOutput which returns
         ↪  inputAmount
        uint256 returnedInputAmount =
↪  pool.sellPoolTokens(poolTokensToSell);
```

```
        uint256 actualWeth = weth.balanceOf(user) - wethBefore;
        uint256 actualPoolTokensSpent = poolTokensBefore -
↪  poolToken.balanceOf(user);

        // Bug: sellPoolTokens treats poolTokenAmount as WETH output, not
        ↪  pool token input
        // So it tries to get 10e18 WETH, requiring way more pool tokens
        ↪  than 10e18
        assertGt(actualPoolTokensSpent, poolTokensToSell, "BUG: User spent
        ↪  more tokens than intended");
        assertNotEq(actualWeth, expectedWeth, "BUG: Wrong WETH amount
        ↪  received");

        // Demonstrate correct behavior: recalculate expected with current
        ↪  reserves
        // (reserves changed after the incorrect swap, but we can still
        ↪  show the difference)
        uint256 correctWeth = pool.getOutputAmountBasedOnInput(
            poolTokensToSell,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );

        console.log("Correct WETH:", correctWeth);

        // The actual WETH received should not match what swapExactInput
        ↪  would give
        // (Note: actualWeth is from the buggy swap, so it's trying to get
        ↪  poolTokensToSell WETH)
        // We expect actualWeth to be close to poolTokensToSell (the bug),
        ↪  not to correctWeth
        assertApproxEqRel(actualWeth, poolTokensToSell, 1e15, "BUG:
        ↪  sellPoolTokens got poolTokenAmount as WETH");
        assertNotEq(actualWeth, correctWeth, "BUG CONFIRMED: sellPoolTokens
        ↪  uses wrong function");
        vm.stopPrank();
    }
```

Recommended Mitigation:

Modify the implementation to utilize `swapExactInput` in place of `swapExactOutput`. Be aware that this modification necessitates updating the `sellPoolTokens` function signature to include an additional parameter (specifically, `minWethToReceive`, which will be forwarded to `swapExact-`

Input).

```
    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));
    }
```

Furthermore, introducing a deadline parameter would be advisable, given that the function presently lacks deadline enforcement, potentially leaving users vulnerable to MEV exploitation.

[H-4] Additional tokens distributed to users after each `swapCount` in `TSwapPool::_swap` violates the protocol's `x * y = k` invariant

Description: The protocol maintains a rigid mathematical relationship expressed as `x * y = k`, where:

- x: Represents the pool token balance
- y: Represents the WETH balance
- k: Represents the constant product of both balances

This relationship requires that when balances change within the protocol, the product of both amounts must stay constant, maintaining the value k. Unfortunately, this fundamental relationship is violated by the bonus reward system implemented in the _swap function, which will ultimately result in the gradual depletion of protocol reserves.

The code segment below is the source of this problem.

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

Impact: An attacker could systematically deplete protocol reserves by executing numerous swaps and accumulating the bonus rewards issued by the protocol.

In essence, the protocol's fundamental mathematical relationship has been compromised.

Proof of Concept:

1. An attacker executes 10 swaps and accumulates the bonus reward of `1_000_000_000_000_000_000` tokens
2. The attacker proceeds to perform additional swaps until protocol reserves are completely exhausted

Proof Of Code

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

```
    /**
     * @notice Proof of Concept for H-4: Extra tokens distributed after swapCount breaks x * y = k invariant
     * @dev This test demonstrates that the bonus reward mechanism violates the constant product formula.
     *      After 9 swaps, the 10th swap triggers a bonus reward, causing the pool's WETH balance
     *      to decrease by more than just the swap amount, breaking the invariant.
     */
    function testInvariantBroken() public {
        // Setup: Initialize pool with liquidity
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        // Define swap parameters: user wants to receive 0.1 WETH per swap
        uint256 desiredWethOutput = 1e17;

        // Prepare user with sufficient tokens
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);

        // Execute 9 swaps to build up swap_count toward SWAP_COUNT_MAX (10)
        // These swaps should not trigger the bonus reward yet
        for (uint256 i = 0; i < 9; i++) {
            pool.swapExactOutput(poolToken, weth, desiredWethOutput, uint64(block.timestamp));
        }
```

```
        // Capture the pool's WETH balance before the 10th swap
        // This swap will trigger the bonus reward mechanism
        uint256 wethBalanceBefore = weth.balanceOf(address(pool));

        // Calculate expected change: should only decrease by the swap
        ↪  amount (desiredWethOutput)
        // The invariant x * y = k requires that balance changes follow the
        ↪  constant product formula
        int256 expectedWethDecrease = -int256(desiredWethOutput);

        // Execute the 10th swap, which triggers the bonus reward
        pool.swapExactOutput(poolToken, weth, desiredWethOutput,
↪  uint64(block.timestamp));
        vm.stopPrank();

        // Verify the actual change in WETH balance
        uint256 wethBalanceAfter = weth.balanceOf(address(pool));
        int256 actualWethChange = int256(wethBalanceAfter) -
↪  int256(wethBalanceBefore);

        // Assertion: The actual change should equal the expected change
        // If this fails, it proves the bonus reward breaks the invariant
        // The actual change will be more negative than expected due to the
        ↪  bonus reward
        assertEq(actualWethChange, expectedWethDecrease, "Invariant broken:
        ↪  bonus reward causes unexpected balance change");
    }
```

Recommended Mitigation: Eliminate the bonus reward system. Should you wish to retain this functionality, adjust calculations to preserve the `x * y = k` protocol invariant. Another option is to reserve tokens using the same methodology employed for fee management.

```
-         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` lacks deadline validation, allowing transactions to execute beyond intended deadline

Description: Although the `deposit` function includes a deadline parameter that the documentation defines as "The deadline for the transaction to be completed by", this parameter is completely ignored in the implementation. Consequently, liquidity provision operations may be processed during unfavorable market conditions, potentially at times when deposit rates are suboptimal.

Impact: Users may submit transactions during adverse market conditions for deposits, despite providing a deadline parameter that should prevent such execution.

Proof of Concept: The `deadline` parameter remains unvalidated within the function.

Recommended Mitigation: Implement the following modification 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)
    {
```

[M-2] Rebase, fee on transfer, and ERC777 tokens break protocol invariant

Description: The protocol relies on `balanceOf()` to determine reserves and assumes standard ERC20 transfer behavior. However, special token types break these assumptions: fee-on-transfer tokens cause the pool to receive less than `inputAmount` due to transfer fees, rebase tokens can change balances between reserve reads and transfers, and ERC777 tokens execute hooks during transfers that can modify balances or cause reentrancy. The protocol calculates reserves before transfers (lines 310-311, 348-349) but actual balances after transfers may differ, breaking the `x * y = k` invariant.

Impact: The protocol invariant is violated when using non-standard tokens, causing users to receive incorrect amounts, potential fund loss, and reentrancy vulnerabilities.

Proof of Concept:

1. Fee-on-transfer: Pool calculates it needs 100 tokens, user transfers 100 tokens, but pool only receives 95 tokens (5% fee). Actual reserves are 5 tokens less than calculated, breaking the invariant.
2. Rebase: Pool reads reserves as 1000 tokens, token rebases increasing balances by 10% before swap executes. Pool now has 1100 tokens but calculations assumed 1000, causing incorrect swap amounts.
3. ERC777: Transfer hooks modify balances or cause reentrancy, making protocol state inconsistent.

Recommended Mitigation:

For fee-on-transfer tokens, calculate actual received amount after transfer:

```
+       uint256 balanceBefore = inputToken.balanceOf(address(this));
        inputToken.safeTransferFrom(msg.sender, address(this),
↪  inputAmount);
+       uint256 balanceAfter = inputToken.balanceOf(address(this));
+       uint256 actualInputReceived = balanceAfter - balanceBefore;
+       // Use actualInputReceived instead of inputAmount
```

For rebase and ERC777 tokens, either read reserves immediately before and after transfers, or explicitly disallow these token types. Consider maintaining internal reserve accounting instead of relying solely on `balanceOf()`.

Low

[L-1] `TSwapPool::LiquidityAdded` event parameters are arranged incorrectly

Description: Within the `TSwapPool::_addLiquidityMintAndTransfer` function, the `LiquidityAdded` event emits values in the wrong sequence. The `wethToDeposit` value belongs in the second parameter slot, while `poolTokensToDeposit` should occupy the third parameter position.

Impact: The event emits data in the wrong order, which may cause off-chain systems that rely on this event to function incorrectly.

Recommended Mitigation:

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

[L-2] `TSwapPool::swapExactInput` returns default value, producing inaccurate return data

Description: The `swapExactInput` function should return the precise quantity of tokens acquired by the caller. Despite declaring a named return variable `output`, this variable is never populated with a value, and the function does not include an explicit return statement.

Impact: The function will consistently return 0, providing misleading information to callers.

Recommended Mitigation:

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

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

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

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

Informationals

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` error is unused and should be eliminated

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

[I-2] Missing zero address validation

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

```
        i_wethToken = wethToken;
    }
```

[I-3] `PoolFactory::createPool` ought to utilize `.symbol()` rather than `.name()`

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

[I-4] Events lack `indexed` field declarations

Indexing event fields enhances accessibility for off-chain applications that process event logs. It is important to recognize that each indexed field increases gas consumption during event emission, so maximizing the number of indexed fields (three per event) may not always be optimal. Events containing three or more fields should index three fields when gas costs are not a primary consideration. For events with fewer than three fields, all fields should be indexed.

- Found in src/TSwapPool.sol: Line: 44
- Found in src/PoolFactory.sol: Line: 37
- Found in src/TSwapPool.sol: Line: 46
- Found in src/TSwapPool.sol: Line: 43