TSwap protocol Report

Version 1.0

Ranir

July 3, 2025

Protocol Audit Report

Ranir

July,2 2025

Prepared by: Cyfrin Lead Auditors: - Ranir

Table of Contents

Protocol Summary

TSwap is a decentralized protocol (DEX) that allows users to exchange tokens with one another at a fair price, without needing a central authority.Unlike traditional order book exchanges, TSwap uses an Automated Market Maker (AMM) model based on liquidity pools (similar to Uniswap).The main contract, PoolFactory, is responsible for creating pools via TSwapPool contracts — each representing an exchange between an ERC20 token and WETH.With enough pools deployed, users can seamlessly swap between ERC20 tokens by routing through WETH.

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

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	5
Total	12

Findings

High

[H-1] Incorrect fee scaling in `TSwapPool::getInputAmountBasedOnOutput` leads to excessive token withdrawal from users

Description: The `TSwapPool::getInputAmountBasedOnOutput`function is responsible for computing how many input tokens a user must provide to receive a specific `outputAmount`. However, the current fee calculation mistakenly uses a scaling factor of 10_000 instead of 1_000, which is inconsistent with the 0.3% fee logic commonly used in AMM formulas.

Impact: This miscalculation causes the protocol to withdraw more tokens than necessary, effectively overcharging users and reducing trust. It may also break integrations or price assumptions on frontends.

Proof of Concept: Run the following code in your in your tes file

```
function test_FaultyFeeCalculationOverchargesUser() public {
    uint256 inputReserves = 1000;
    uint256 outputReserves = 1000;
    uint256 outputAmount = 100;

    // actual version of the bug
    uint256 overchargedInput = pool.getInputAmountBasedOnOutput(
        outputAmount,
        inputReserves,
        outputReserves
    );

    // Corrected version with 0.3% fee
    uint256 expectedInput = (inputReserves * outputAmount * 1000) /
                            ((outputReserves - outputAmount) * 997);
```

```
16
17      // Verification: The bug makes you pay much more than expected
18      assertEq(overchargedInput, expectedInput * 2); // example: at least
            2x more
19      emit log_named_uint("Overcharged input", overchargedInput);
20      emit log_named_uint("Expected input", expectedInput);
21  }
```

Recommended Mitigation:

```
1       function getInputAmountBasedOnOutput(
2           uint256 outputAmount,
3           uint256 inputReserves,
4           uint256 outputReserves
5       )
6           public
7           pure
8           revertIfZero(outputAmount)
9           revertIfZero(outputReserves)
10          returns (uint256 inputAmount)
11      {
12  -       return ((inputReserves * outputAmount) * 10_000) / ((
        outputReserves - outputAmount) * 997);
13  +       return ((inputReserves * outputAmount) * 1_000) / ((
        outputReserves - outputAmount) * 997);
14      }
```

[H-2] Lack of slippage protection in TSwapPool::swapExactOutput causes users to potentially overpay

Description: The function `swapExactOutput` lets users specify the amount of tokens they want to receive, but does not allow them to set a maximum amount they're willing to pay.Unlike `swapExactInput`, which includes `minOutputAmount` to protect users from slippage, `swapExactOutput` has no `maxInputAmount` parameter. As a result, users may unknowingly spend much more than expected, especially under volatile conditions or front-running.

Impact: This exposes users to high slippage and economic loss if the pool state changes between transaction submission and execution. It also opens the door to MEV-style attacks, where bots manipulate reserves to extract more input tokens from users.

Proof of Concept: Add the following code to TSwapPool.t.sol file.

```
1   function test_swapExactOutputWithoutMaxInputCausesOverpayment() public
        {
2       // Simulate a pool with initial price: 1 WETH = 1_000 USDC
3       uint256 wethReserves = 100 * 1e18;   // 100 WETH
4       uint256 usdcReserves = 100_000 * 1e6; // 100,000 USDC (6 decimals)
```

```
5
6       // The User want's 1 WETH
7       uint256 outputAmount = 1e18;
8
9       // Expected price,  1,000 USDC
10      uint256 expectedInput = (usdcReserves * outputAmount * 1000) /
11                              ((wethReserves - outputAmount) * 997);
12
13      // A large transaction changes the market , 1 WETH = 10,000 USDC
14      wethReserves = 100 * 1e18;
15      usdcReserves = 1_000_000 * 1e6;
16
17      // We recalculate the input after slippage
18      uint256 actualInput = (usdcReserves * outputAmount * 1000) /
19                            ((wethReserves - outputAmount) * 997);
20
21      // The protocol accepts the transaction without maxInput => big
            overpayment
22      assertGt(actualInput, expectedInput * 5); //*the user pays 5x more
            than expected
23
24      emit log_named_uint("Expected Input (before slippage)",
            expectedInput);
25      emit log_named_uint("Actual Input (after slippage)", actualInput);
26  }
```

Recommended Mitigation: Update the function signature:

```
1       function swapExactOutput(
2           IERC20 inputToken,
3   +       uint256 maxInputAmount,
4   .
5   .
6   .
7           inputAmount = getInputAmountBasedOnOutput(outputAmount,
                inputReserves, outputReserves);
8   +       if(inputAmount > maxInputAmount){
9   +           revert();
10  +       }
11          _swap(inputToken, inputAmount, outputToken, outputAmount)
12                              );
```

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

Description: The `sellPoolTokens` function is designed to allow users to redeem their pool tokens in exchange for WETH. The user specifies the number of pool tokens to sell via the poolTokenAmount parameter. However, the function contains a flaw in how it calculates the corresponding WETH output

amount. This miscalculation leads to incorrect swap results, potentially causing users to receive significantly less WETH than expected or disrupting the pool's balance.

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

Proof of Concept:add this function to TSwapPool.t.sol

```
function test_sellPoolTokens_misuses_swapExactOutput() public {
    uint256 poolTokenAmount = 1000e18;


    i_poolToken.mint(address(this), poolTokenAmount);
    i_poolToken.approve(address(tswapPool), type(uint256).max);


    uint256 wethBefore = weth.balanceOf(address(this));

    uint256 wethReceived = tswapPool.sellPoolTokens(poolTokenAmount);

    uint256 wethAfter = weth.balanceOf(address(this));
    uint256 actualWeth = wethAfter - wethBefore;

    assertLt(actualWeth, poolTokenAmount / 2);

    console2.log("Expected ~", poolTokenAmount);
    console2.log("Actual WETH received:", actualWeth);
}
```

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));
    }
```

[H-4] In TSwapPool::_swap, the bonus tokens granted to users every swapCount iterations violate the core x * y = k invariant of the constant product formula.

Description: The protocol enforces a strict invariant based on the constant product formula x * y = k, where: 1. x is the balance of the pool token, 2. y is the balance of WETH, 3. k is the constant product that ensures price stability. This invariant ensures that any swap operation preserves the proportional relationship between token reserves. However, in the `_swap` function, an additional incentive (extra tokens) is granted to users every swapCount iterations. This external injection of value breaks the x * y = k invariant, ultimately leading to a depletion of protocol reserves over time.The follow block of code is responsible for the issue.

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

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.

Proof of Concept: Add the following test to the TSwapPool.t.sol 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 until all the protocol funds are drained.

Proof Of Code

```
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.mint(user, 1_000e18);

        poolToken.approve(address(pool), type(uint256).max);
        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));
```

```
18          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
19          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
20          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
21          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
22          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
23
24          int256 startingX = int256(weth.balanceOf(address(pool)));
25          int256 expectedDeltaX = int256(outputWeth) * -1;
26
27          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
28
29          vm.stopPrank();
30
31          uint256 endingX = weth.balanceOf(address(pool));
32          int actualDeltaX = int256(endingX) - int256(startingX);
33
34          assertEq(actualDeltaX, expectedDeltaX);
35      }
```

Recommended Mitigation: Remove the extra incentive mechanism. If you want to keep this in, we should account for the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

```
1 -         swap_count++;
2 -         // Fee-on-transfer
3 -         if (swap_count >= SWAP_COUNT_MAX) {
4 -             swap_count = 0;
5 -             outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
6 -         }
```

Medium

[M-1] `TSwapPool::deposit` is missing a deadline check, allowing transactions to execute after expiration

Description: The `deposit` function takes a deadline parameter meant to restrict execution to a specific timeframe. However, this value is never enforced. As a result, deposits may be executed after the intended expiration, exposing users to unfavorable rates or slippage due to market shifts.

Impact: Without enforcing the deadline, users are exposed to front-running or delayed execution,

allowing deposits to occur under unfavorable market conditions despite specifying a deadline for protection.

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

Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning. –> src/TSwapPool.sol:96:9: | 96 | uint64 deadline | ^^^^^^^^^^^^^^^^

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

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

Low

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

Description: The `LiquidityAdded` event is emitted with its arguments in an inconsistent order. According to the function logic and parameter naming, `wethToDeposit` should appear as the second argument and `poolTokensToDeposit` as the third. Swapping them creates a mismatch between emitted logs and expected values by off-chain systems.

Impact: Incorrect event structure can break off-chain integrations (indexers, frontends, analytics), leading to misinterpreted liquidity data or silent failures.

Proof of Concept: The LiquidityAdded event in TSwapPool is declared as:

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

However, it is emitted with arguments in the wrong order:

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

As a result`poolTokensToDeposit` is mistakenly logged as `wethDeposited`and `wethToDeposit` is mistakenly logged as `poolTokensDeposited`.

Recommended Mitigation: Reorder the parameters in the event emission to match the expected layout:

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

[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(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);
}
}
```

Informational

[I-1] `PoolFactory::PoolFactory___PoolDoesNotExist` appears to be unused and should be removed to improve code clarity and maintainability.

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

[I-2] PoolFactory::constructor lacks a zero address check

Description: The constructor does not validate the `wethToken` address, allowing potential initialization with the zero address (address(0)), which can lead to unexpected behavior or broken functionality.
Recommended Mitigation: Add an explicit check to ensure that the provided wethToken is not the zero address.

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

[I-3] `PoolFactory::createPool` uses .name() instead of .symbol() for liquidity token symbol

Recommended Mitigation: Replace .name() with .symbol() for better alignment with ERC-20 naming conventions:

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

[I-4] `TSwapPool::constructor` Lacking zero address check - wethToken & poolToken

Recommended Mitigation:

```
constructor(
    address poolToken,
    address wethToken,
    string memory liquidityTokenName,
    string memory liquidityTokenSymbol
)
    ERC20(liquidityTokenName, liquidityTokenSymbol)
{
+   if(wethToken || poolToken == address(0)){
+       revert();
+   }
    i_wethToken = IERC20(wethToken);
```

```
13         i_poolToken = IERC20(poolToken);
14     }
```

[I-5] `TSwapPool` events should use indexed keywords to improve query efficiency

Description: The Swap event currently indexes only the swapper address. However, both `tokenIn` and `tokenOut` are also valuable filtering criteria for off-chain consumers (e.g., indexers, dashboards, analytics). Without indexed, it becomes costly or impractical to search logs by token addresses.

Recommended Mitigation: Index both tokenIn and tokenOut to enhance on-chain event filtering and log querying performance:

```
1 - event Swap(address indexed swapper, IERC20 tokenIn, uint256
      amountTokenIn, IERC20 tokenOut, uint256 amountTokenOut);
2 + event Swap(address indexed swapper, IERC20 indexed tokenIn, uint256
      amountTokenIn, IERC20 indexed tokenOut, uint256 amountTokenOut);
```