TSwap Protocol Audit Report

Cyfrin.io, 0xOwain

September 2, 2025

TSwap Protocol Audit Report

Version 1.0

Cyfrin.io

September 2, 2025

TSwap Protocol Audit Report

Cyfrin.io, 0xOwain

September 2, 2025

Prepared by: Cyfrin Lead Auditors: - 0xOwain

Table of Contents

Protocol Summary

TSwapPool is a custom token swap pool protocol, designed to facilitate token-to-token swaps without relying on external DEXs like Uniswap. It allows users to: - Deposit liquidity in the form of two ERC-20 tokens (e.g., WETH and a pool token), - Mint pool tokens representing their share of the liquidity pool, - Swap tokens using constant product AMM logic (x * y = k) via swapExactInput and swapExactOutput functions, - Withdraw liquidity while reclaiming fees earned, - Sell pool tokens directly for WETH via a helper function sellPoolTokens.

The pool maintains internal reserves of the two tokens and calculates swap amounts using deterministic formulas, similar to Uniswap v2. A swap fee is charged on each trade, and the contract includes functions for fee collection and invariant enforcement.

The design prioritises simplicity and fast execution, but assumes ideal behaviour from ERC-20 tokens and omits several safety features (e.g., slippage protection, token compatibility checks), leading to several critical and medium-severity vulnerabilities.

Disclaimer

The 0xOwain 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
Likelihood	High	H	H/M	M
	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

Target Contracts:

`TSwapPool.sol` — core logic for token swaps, deposits, withdrawals, and fee management.

`PoolFactory.sol` — factory contract responsible for deploying and managing instances of `TSwapPool`.

Solidity Version: `0.8.20`

Commit Hash: c7ddc8e96d9a5656c9ca1585c5d57532121fca86

cloc:

File blank comment code

src/TSwapPool.sol 43 169 228 src/PoolFactory.sol 11 36 35 —
SUM: 54 205 263 —

Roles

Role Responsibility

Liquidity Provider: Supplies both tokens (e.g., WETH and pool token) into the pool and receives LP tokens in return. Can later withdraw their share along with earned fees.

Swapper / Trader: Uses `swapExactInput` or `swapExactOutput` to exchange between the two tokens held by the pool. Pays swap fees and must manage slippage.

Protocol / Admin: (Assumed implicit) May be responsible for deploying the pool contracts, setting initial parameters, and handling future upgrades or fee logic if present.

Pool Token Holder: Holds pool tokens and can redeem them for WETH using the `sellPoolTokens` shortcut.

Executive Summary

This report presents the findings from a security review of the TSwapPool smart contract system. The protocol implements a custom liquidity pool and swap mechanism, allowing users to deposit two ERC-20 tokens, mint pool tokens, and perform token swaps using constant product AMM logic (x * y = k). It also offers functionality for fee collection and direct pool token redemption.

The audit uncovered 17 issues in total, including: - 4 High severity issues – affecting core swap logic, fee calculation, and token compatibility assumptions - 2 Medium severity issues – related to slippage protection and misuse of internal logic - 2 Low severity issues – mostly affecting clarity or minor edge-case handling - 9 Informational findings – non-critical, but worth addressing to improve maintainability and safety

Several issues could lead to incorrect pricing, loss of funds, or disrupted protocol behavior if not resolved. Of particular concern is the protocol's assumption that all ERC-20 tokens behave in a standard, 1:1 manner, which opens the door to logic breaks when interacting with fee-on-transfer or rebase tokens.

This audit did not cover economic modeling, price manipulation resistance, or frontend integration. The audit assumes honest and trusted deployment.

The code demonstrates a solid grasp of DeFi primitives, but would benefit from tighter assumptions, safer token handling, and clearer function intent.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	9
Total	**17**

Findings

High

[H-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, operations that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavourable.

Impact: Transactions could be sent when market conditions are unfavourable 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%
     uint256 maximumPoolTokensToDeposit,
     uint64 deadline
 )
     external
+    revertIfDeadlinePassed(deadline)
     revertIfZero(wethToDeposit)
     returns (uint256 liquidityTokensToMint)
 {
```

[H-2] Incorrect fee calculation in `TSwapPool:: getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees

Description: The `getINputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user shjould 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.

Impact: Protocol takes more fees than expected from users.

Proof of Concept:

```
function testFeeCalculationIsIncorrect() public {
        uint256 inputReserve = 10_000 ether;
        uint256 outputReserve = 10_000 ether;
        uint256 desiredOutput = 1 ether;

        uint256 inputWithBug = pool.getInputAmountBasedOnOutput(desiredOutput, inputReserve,

        uint256 correctInput = (inputReserve * desiredOutput * 1000) / ((outputReserve - des

        console.log("With bug:", inputWithBug);
        console.log("Expected:", correctInput);

        assertGt(inputWithBug, correctInput);


    }
```

Recommended Mitigation:

```
function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(outputAmount)
    revertIfZero(outputReserves)
    returns (uint256 inputAmount)
{
    return ((inputReserves * outputAmount) * 1000) / ((outputReserves - outputAmount) * 997)
    }
```

[H-3] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentiall 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 change before the transaction processes, the user could get a much worse swap.

Proof of Concept:

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

        address attacker = makeAddr("attacker");
        poolToken.mint(attacker, 100e18);
        vm.startPrank(attacker);
        poolToken.approve(address(pool), 100e18);

        pool.swapExactInput(poolToken, 100e18, weth, 1e18, uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(user);
        weth.mint(user, 100e18);
        weth.approve(address(pool), 100e18);

        uint256 wethBefore = weth.balanceOf(user);

        pool.swapExactOutput(weth, poolToken, 10e18, uint64(block.timestamp));

        uint256 wethAfter = weth.balanceOf(user);
        uint256 wethSpent = wethBefore - wethAfter;

        console.log("WETH spent: ", wethSpent);

        assertGt(wethSpent, 20e18);
        vm.stopPrank();
    }
```

Recommended Mitigation:

```
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));

    inputAmount = getInputAmountBasedOnOutput(outputAmount, inputReserves, outputReserves);

+   require(inputAmount <= maxInputAmount, "TSwap: too much input required");

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

[H-4] `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 tehy'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:

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

        vm.startPrank(user);
        poolToken.mint(user, 100e18);
        poolToken.approve(address(pool), 100e18);
```

```
        uint256 expectedWethAmount = pool.swapExactInput(
            poolToken,
            100e18,
            weth,
            1,
            uint64(block.timestamp)
        );

        poolToken.mint(user, 100e18);
        poolToken.approve(address(pool), 100e18);

        uint256 actualWethAmount = pool.sellPoolTokens(1e18);

        console.log("Expected WETH: ", expectedWethAmount);
        console.log("Actual WETH from sellPoolTokens: ", actualWethAmount);
        assertNotEq(actualWethAmount, expectedWethAmount, "sellPoolTokens miscalculates outp

        vm.stopPrank();
    }
```

Recommended Mitigation:

```
- function sellPoolTokens(uint256 poolTokenAmount) external returns (uint256 wethAmount) {
-     // pool token -> input
-     // @audit this is wrong!!!!
-     // swapExactInput(minWethToReceive)
-     return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount, uint64(block.timesta
- }

+ function sellPoolTokens(uint256 poolTokenAmount, uint256 minWethAmount) external returns
+     // The user specifies how many pool tokens they want to sell (exact input)
+     // and sets a minimum amount of WETH they'd like to receive (slippage protection).
+     //
+     // Use swapExactInput instead of swapExactOutput to match intent
+     wethAmount = swapExactInput(i_poolToken, poolTokenAmount, i_wethToken, minWethAmount,
+ }
```

[H-5] 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 the two balances

This means that whenever the balances change in the protocol, the ratio between the two amounts should remain constant, hense 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++;
if (swap_count >= SWAP_COUNT_MAX) {
    swap_count = 0;
    outputToken.safeTrtansfer(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 incensive given out by the protocol.

Most simply put, the protocl'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 until all the protocol funds are drained.

Proof of Code

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

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

        uint256 outputWeth = 1e17;

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.timestamp));

        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);

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

```
        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));

        int256 actualDeltaY = int256(endingY) - int256(startingY);
        assertEq(actualDeltaY, expectedDeltaY);
    }
```

Recommended Mitigation: Remove the extra incentive. 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.

```
-    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-2] rebase, fee-on-transfer, and ERC777 tokens break protocol invariant

Description: The protocol assumes that ERC20 token transfers are 1:1 exact meaning the number of tokens sent via `transferFrom` or `transfer` is equal to the amount actually received or deducted.

However, this assumption is invalid for non-standard ERC20 tokens, such as: - Fee-on-transfer (e.g., deduct 1-5% on each transfer) - Rebase tokens (e.g., Ampleforth, which change balances algorithmically) - ERC777 tokens (support hooks that can execute arbitrary logic on transfer)

These tokens can break key assumptions in the TSwapPool, including: - Tokens received are equal to `inputAmount` - `balanceOf` reflects actual changes before and after a transfer - No external logic executes unexpectedly during transfers.

Many key functions (e.g., `swapExactInput`, `deposi`, `withdraw`) assume standard behaviour without verification.

Impact: Failure to account for non-standard token behaviour leads to broken invariants, mispriced swaps, and the potential for manipulation or loss.

Proof of Concept:

```
function testFeeOnTransferTokenBreaksInvariant() public {
    FeeToken feeToken = new FeeToken();

    TSwapPool feePool = new TSwapPool(address(feeToken), address(weth), "LTFee", "LTF");
```

```
        feeToken.mint(liquidityProvider, 100e18);
        weth.mint(liquidityProvider, 100e18);

        vm.startPrank(liquidityProvider);
        feeToken.approve(address(feePool), 100e18);
        weth.approve(address(feePool), 100e18);

        feePool.deposit(100e18, 90e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 actualTokenBalance = feeToken.balanceOf(address(feePool));
        uint256 expected = 100e18;

        console.log("Expected token balance: ", expected);
        console.log("Actual token balance:   ", actualTokenBalance);

        assertLt(actualTokenBalance, expected);
    }
```

Recommended Mitigation: 1. Support all ERC20s Use the balance delta pattern to compute the actual received token amount: `solidity uint256 before = token.balanceOf(address(this)); token.safeTransferFrom(msg.sender, address(this), amount); uint256 received = token.balanceOf(address(this)) - before;` Then use `received` (not `amount`) in all calculations. This ensures the contract uses the real amount received, even if fees were taken or rebase logic triggered.

2. Restrict to safe tokens If supporting non-standard tokens is not a priority, explicitly restrict deposits/swaps to known-safe ERC20s using an allowlist: `solidity require(isSafeToken[token], "Unsupported token");` This prevents users from injecting malicious tokens that can break pool logic.

Low

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

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTransfer` function, it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the third parameter position, whereas the `wethToDeposit` value should go second.

Impact: Event emission is incorrect, leading to off-chain functions potentially malfunctioning.

Recommended Mitigation:

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

Informationals

[I-1] `PoolFactory::PoolFactory_PoolDoesNotExist` is not used and should be removed

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

[I-2] Lacking zero address checks

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

[I-3] `PoolFactory::createPool` should use `.symbol()` instead of `name()`

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

[I-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the 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