Protocol Audit Report

Version 1.0

Cyfrin.io

November 21, 2025

T-Swap Protocol Audit Report

Bree

November 21, 2025

Prepared by: Bree Lead Auditors: - Bree

Table of Contents

    * [H-5] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`

- Lows
    * [L-1] `TSwapPool::LiquidityAdded` event has parameters out of order
    * [L-2] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given
- Informationals
    * [I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed
    * [I-2] Lacking zero address checks
    * [I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`
    * [I-4] Event is missing `indexed` fields

Protocol Summary

T-Swap is a decentralized exchange (DEX) that enables users to permissionlessly swap between two assets using Automated Market Maker (AMM) mechanics instead of an order-book model. Liquidity providers deposit pairs of tokens into a shared pool, and traders swap against these pooled reserves at a price determined algorithmically by the pool's reserve ratio.

Disclaimer

The Bree 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

		Impact		
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

- Commit Hash: 1ec3c30253423eb4199827f59cf564cc575b46db
- In Scope:

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum

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

This audit evaluated the security, correctness, and economic safety of the T-Swap protocol — a permissionless Automated Market Maker (AMM) inspired by Uniswap. The goal of T-Swap is to allow users to swap assets at fair market rates using liquidity pools instead of an order book.

Issues found

Severity	Number of issues found
High	5
Medium	0
Low	2
Info	4
Total	11

Findings

High

[H-1] Missing Deadline Validation in `TSwapPool::deposit` Function

Description: The `deposit` function accepts a `deadline` parameter but fails to validate it against the current `block timestamp`. This allows pending transactions to be executed long after the user's intended expiration time, potentially resulting in unfavorable trade conditions due to price slippage or market movements.

Impact: Users may receive significantly worse pricing than expected if their transaction is mined after market conditions have changed. In extreme cases, this could lead to substantial financial losses when depositing liquidity during periods of high volatility.

Proof of Concept: The `deadline` parameter is unused

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)
```

[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 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.

```
return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
    outputAmount) * 997);
```

Impact: Protocol takes more fees than expected from users

Proof of Concept:

- Protocol charged expectedBuggy
- Which is 10× more than mathematically required
- Demonstrating loss of user funds due to wrong 10_000 multiplier
  Here is a PoC test that proves the excess fee issue.

```
function testExcessFeesDueToBadMultiplier() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();
        poolToken.mint(user, 5e18);

        uint256 outputAmount = 1e18;
        uint256 inputReserves = poolToken.balanceOf(address(pool));
        uint256 outputReserves = weth.balanceOf(address(pool));

        uint256 expectedInputCorrect =
            ((inputReserves * outputAmount) * 1_000) / ((outputReserves
                - outputAmount) * 997);
        uint256 expectedInputBuggy = ((inputReserves * outputAmount) *
            10_000) / ((outputReserves - outputAmount) * 997);

        vm.startPrank(user);
        poolToken.approve(address(pool), expectedInputBuggy);
        uint256 userBalanceBefore = poolToken.balanceOf(user);
        pool.swapExactOutput(poolToken, weth, outputAmount, uint64(
            block.timestamp));
        vm.stopPrank();

        uint256 tokensSpent = userBalanceBefore - poolToken.balanceOf(
            user);

```

```
25          console.log("Correct required input:   ", expectedInputCorrect)
                ;
26          console.log("Buggy required input:     ", expectedInputBuggy);
27          console.log("Actual input taken:       ", tokensSpent);
28
29          assertEq(tokensSpent, expectedInputBuggy);
30          assertGt(tokensSpent, expectedInputCorrect);
31
32          assertGt(tokensSpent - expectedInputCorrect, 0.5e18);
33      }
```

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
13 -         return ((inputReserves * outputAmount) * 10_000) / ((
       outputReserves - outputAmount) * 997);
14 +         return ((inputReserves * outputAmount) * 1_000) / ((
       outputReserves - outputAmount) * 997);
15     }
```

[H-3] 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 change before the transcation processes, the user could 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 a maxInput amount 4. As the transcation 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 sent the protocol 10,000 usdc instead of

the expected 1,000 usdc

PoC test demonstrating the lack of slippage protection in swapExactOutput()

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

        poolToken.mint(user, 100e18);

        vm.startPrank(user);
        poolToken.approve(address(pool), 20 ether);
        vm.stopPrank();

        uint256 outputAmount = 1e18;

        uint256 inputReservesBefore = poolToken.balanceOf(address(pool)
            );
        uint256 outputReservesBefore = weth.balanceOf(address(pool));

        uint256 expectedInput =
            ((inputReservesBefore * outputAmount) * 1_000) / ((
                outputReservesBefore - outputAmount) * 997);
        console.log("Expected input (current reserves):", expectedInput
            );

        vm.startPrank(liquidityProvider);
        pool.approve(address(pool), 50e18);
        pool.withdraw(50e18, 50e18, 50e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 inputReservesAfter = poolToken.balanceOf(address(pool))
            ;
        uint256 outputReservesAfter = weth.balanceOf(address(pool));

        console.log("New input reserves:", inputReservesAfter);
        console.log("New output reserves:", outputReservesAfter);

        uint256 userBalanceBefore = poolToken.balanceOf(user);

        vm.prank(user);
        uint256 inputSpent = pool.swapExactOutput(poolToken, weth,
            outputAmount, uint64(block.timestamp));

        uint256 userBalanceAfter = poolToken.balanceOf(user);

        console.log("Input spent by user:", inputSpent);
        console.log("User balance change:", userBalanceBefore -
```

```
            userBalanceAfter);
43
44          assertGt(inputSpent, expectedInput, "User spent more than
               expected due to no slippage protection");
45      }
```

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

```
1  function swapExactOutput(
2          IERC20 inputToken,
3          IERC20 outputToken,
4  +       maxInputAmount,
5          uint256 outputAmount,
6          uint64 deadline
7      )
8          public
9          revertIfZero(outputAmount)
10         revertIfDeadlinePassed(deadline)
11         returns (uint256 inputAmount)
12     {
13         uint256 inputReserves = inputToken.balanceOf(address(this));
14         uint256 outputReserves = outputToken.balanceOf(address(this));
15
16         inputAmount = getInputAmountBasedOnOutput(outputAmount,
              inputReserves, outputReserves);
17 +        if(inputAmount > maxInputAmount) {
18 +            revert();
19 +        }
20         _swap(inputToken, inputAmount, outputToken, outputAmount);
21     }
```

[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 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 instead of `swapExactInput`. Because the users specify the exact amount of input tokens not the output.

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

Proof of Concept:

1. sellAmount param passed: 1e18. This is the user's intention: "I want to sell 1 pool token."

2. Expected WETH if using swapExactInput. This is how much WETH the user should have received for selling 1 pool token if the function used swapExactInput. Slightly less than 1 because of the pool's 0.3% fee and the current reserves.
3. Because sellPoolTokens calls swapExactOutput instead of swapExactInput, the argument sellAmount is treated as desired WETH output, not input. The pool calculates how many pool tokens are required to get 1 WETH using the AMM formula. It turns out the user has to spend ~10.13 pool tokens to get 1 WETH.
4. The user received exactly the sellAmount as WETH because swapExactOutput interprets it as requested output, not input.

PoC demonstrates the bug in sellPoolTokens

```
function testSellPoolTokensIncorrectLogic() 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 sellAmount = 1e18; // user intends to sell 1 pool token
        poolToken.mint(user, 10e18);

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

        uint256 userPoolBefore = poolToken.balanceOf(user);
        uint256 userWethBefore = weth.balanceOf(user);

        uint256 inputReserves = poolToken.balanceOf(address(pool));
        uint256 outputReserves = weth.balanceOf(address(pool));
        uint256 expectedWethIfUsingSwapExactInput =
            (sellAmount * outputReserves * 997) / (inputReserves * 1000
                + sellAmount * 997);

        console.log("Expected WETH if using swapExactInput (correct):",
            expectedWethIfUsingSwapExactInput);

        vm.prank(user);
        uint256 returned = pool.sellPoolTokens(sellAmount);

        uint256 userPoolAfter = poolToken.balanceOf(user);
        uint256 userWethAfter = weth.balanceOf(user);

        uint256 poolTokensSpent = userPoolBefore - userPoolAfter;
        uint256 wethReceived = userWethAfter - userWethBefore;

        console.log("sellAmount param passed:         ", sellAmount);
```

```
35          console.log("poolTokens actually spent:      ", poolTokensSpent
                );
36          console.log("weth received by user:          ", wethReceived);
37          console.log("value returned by sellPoolTokens:", returned);
38
39          assertEq(
40              wethReceived, sellAmount, "Bug: user received WETH equal to
                    the sellAmount param (interpreted as output)"
41          );
42          assertEq(
43              returned, poolTokensSpent, "Returned value equals pool
                   tokens spent (swapExactOutput returns inputAmount)"
44          );
45          assertTrue(
46              wethReceived != expectedWethIfUsingSwapExactInput,
47              "WETH received differs from expected swapExactInput result"
48          );
49          assertGe(poolTokensSpent, sellAmount);
50          assertGt(poolTokensSpent, sellAmount);
51      }
```

Recommended Mitigation:

Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note: this will require also changing the `sellPoolTokens` function to accept a new parameter i.e `minWethToReceive` to be passed to `swapExactInput`

```
1   function sellPoolTokens(
2       uint256 poolTokenAmount,
3   +   uint256 minWethToReceive
4       ) external returns (uint256 wethAmount) {
5
6   -       return swapExactOutput(i_poolToken, i_wethToken,
        poolTokenAmount, uint64(block.timestamp));
7   +       return swapExactOutput(i_poolToken, poolTokenAmount,
        i_wethToken, minWethToReceive, uint64(block.timestamp));
8       }
```

Additionally, it might be good we add a deadline to the function, as there's currently no deadline.

[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, hence the `k`. However, this is broken due to extra incentive in the `_swap`

function. Meaning that over time the protocol funds will be drained. The following block of code is responsible:

```
        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. The protocol core invariant is broken.s

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.

The PoC below demonstrates the bug in `TSwapPool::_swap`

```
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)));
```

```
24          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
25
26          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
27          vm.stopPrank();
28
29          uint256 endingY = weth.balanceOf(address(pool));
30          int256 actualDeltaY = int256(endingY) - int256(startingY);
31
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

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

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

Lows

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

Description: The `LiquidityAdded` event is emitted with parameters in the wrong order. The event definition expects `(address liquidityProvider, uint256 wethDeposited, uint256 poolTokensDeposited)` but the emission swaps the `wethDeposited` and `poolTokensDeposited` values, providing incorrect information to off-chain monitors and users.

Impact: External systems, frontends, and users monitoring events will receive inaccurate data about deposit amounts. This could lead to incorrect tracking, reporting, and display of liquidity provision activities, causing confusion and potential mismanagement of position tracking.

Proof of Concept:

```
1 event LiquidityAdded(address indexed liquidityProvider, uint256
     wethDeposited, uint256 poolTokensDeposited);
2
3 emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
```

Recommended Mitigation:

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

```
2 + 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.

Proof of Concept: - The protocol does perform the swap correctly - The user actually receives tokens - But the return value from swapExactInput() is always zero because: - The named return variable output is NEVER assigned - There is NO explicit return output statement

PoC test that demonstrates the issue

```
1  function testSwapExactInputAlwaysReturnsZero() public {
2          vm.startPrank(liquidityProvider);
3          weth.approve(address(pool), 100e18);
4          poolToken.approve(address(pool), 100e18);
5          pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
6          vm.stopPrank();
7
8          vm.startPrank(user);
9          poolToken.approve(address(pool), 10e18);
10
11         uint256 balanceBefore = weth.balanceOf(user);
12         uint256 returned = pool.swapExactInput(
13             poolToken,
14             10e18, // input spent
15             weth,
16             1e18, // minimum output
17             uint64(block.timestamp)
18         );
19         uint256 balanceAfter = weth.balanceOf(user);
20
21         uint256 actualReceived = balanceAfter - balanceBefore;
22
23         console.log("Returned value:     ", returned);
24         console.log("Actual tokens out:  ", actualReceived);
25
26         assertEq(returned, 0, "swapExactInput should return 0 due to
               bug");
27         assertGt(actualReceived, 0, "User actually receives tokens");
28     }
```

Recommended Mitigation:

```
function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        // this hould be external if not called by internal functions
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        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) {
            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` is not used and should be removed

Recommended Mitigation:

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

[I-2] Lacking zero address checks

Recommended Mitigation:

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

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

Recommended Mitigation:

```
- 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