Ramil Must

Protocol Audit Report

Version 1.0

Ramil Mustafin

June 17, 2025

Protocol Audit Report

Ramil Mustafin

June 17, 2025

Prepared by: Ramil Mustafin

Table of Contents

- Medium
  * [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after deadline
- Low
  * [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_PoolDoesNotExist` is not used and should be removed
  * [I-2] `PoolFactory::constructor` lacking zero address check
  * [I-3] `PoolFactory::createPool` should use .symbol() instead of .name()
  * [I-4] `TSwapPool::constructor` Lacking zero address check - wethToken & poolToken
  * [I-5] `TSwapPool` events should be indexed

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.

Disclaimer

Ramil Mustafin 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 security researcher 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

The findings described in this document correspond to the following commit hash: Commit Hash:

```
e643a8d4c2c802490976b538dd009b351b1c8dda
```

Scope

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

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

The audit revealed 12 vulnerabilities, including 4 categories of High, which affect commission calculation, slippage protection and pool liquidity preservation. The most critical bugs allow users to overpay during exchanges or completely deplete the protocol reserves, violating the invariant x - y = k. It is recommended to fix logical errors in swapExactOutput, sellPoolTokens, commission calculation, and reward mechanism, and then re-audit.

Slither and aderyn were used during security-research process.

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	5
Gas	0
Total	12

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput`

Description: The `getInputAmountBasedOnOutput` is intended to calculate the amount of tokens a user should deposit given an amount of tokens 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: This excess user fee can be extracted by the providers. Paste this code into your TSwapPool.t.sol to test it.

Proof of Code

```

    function testSwapExactOutput() public {

        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit(initialLiquidity, 0, initialLiquidity, uint64(
            block.timestamp));
        vm.stopPrank();

        address secUser = makeAddr("secUser");
        uint256 poolTokensToSell = 11e18;
        poolToken.mint(secUser, poolTokensToSell);

        vm.startPrank(secUser);
        poolToken.approve(address(pool), poolTokensToSell);
        pool.swapExactOutput(poolToken, weth, 1 ether, uint64(block.
            timestamp));

        assertLt(poolToken.balanceOf(secUser), 1 ether);
        vm.stopPrank();

        vm.startPrank(liquidityProvider);
        pool.withdraw(pool.balanceOf(liquidityProvider), 1, 1, uint64(
            block.timestamp));

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

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-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 change before the transaction 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 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 sent the protocol 10,000 USDC instead of the expected 1,000 USDC

Paste this code into your TSwapPool.t.sol to test it.

Proof of Code

```
function testSlippageProtectionInSwapExactOutput() public {
        uint256 desiredWeth = 1e18;

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 0, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 expectedInput = pool.getInputAmountBasedOnOutput(
            desiredWeth,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );

        console.log("expectedInput", expectedInput/10e18, "WETH");
        address unprotectedUser = makeAddr("unprotectedUser");

        poolToken.mint(address(pool), 900e18);

        uint256 newInput = pool.getInputAmountBasedOnOutput(
            desiredWeth,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );
        console.log("newInput", newInput/10e18, "WETH");

        uint256 amountToMint = newInput * 2;
        poolToken.mint(unprotectedUser, amountToMint);
        vm.prank(unprotectedUser);
        poolToken.approve(address(pool), amountToMint);

        vm.startPrank(unprotectedUser);
        uint256 poolTokenBefore = poolToken.balanceOf(unprotectedUser);
        pool.swapExactOutput(poolToken, weth, desiredWeth, uint64(block
            .timestamp));
        uint256 poolTokenAfter = poolToken.balanceOf(unprotectedUser);
        uint256 actualInput = poolTokenBefore - poolTokenAfter;
        console.log("actualInput", actualInput/10e18, "WETH");
        vm.stopPrank();

        assertLt(expectedInput * 2, actualInput);

        assertEq(weth.balanceOf(unprotectedUser), desiredWeth);

    }
```

Recommended Mitigation: We should include a `maxInputAmount` so the user only 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 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: Paste this code into your TSwapPool.t.sol to test it.

Proof of Code

```
  function testSellPoolTokens() public {
        uint256 desiredPoolTokens = 10e18;

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 0, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        address secUser = makeAddr("secUser");
        vm.startPrank(secUser);
        poolToken.mint(secUser, desiredPoolTokens);
        poolToken.approve(address(pool), desiredPoolTokens);
        vm.expectRevert();
        pool.sellPoolTokens(desiredPoolTokens);
        vm.stopPrank();
    }
```

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 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 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 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.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 collect 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 outputWethAmount = 1e17;

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

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
```

~~~
        for (uint256 i = 0; i < 10; i++) {
            pool.swapExactOutput(poolToken, weth, outputWethAmount, uint64(block.timestamp));
        }
        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);
        assertEq(actualDeltaY, expectedDeltaY);
}

</details>

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

```diff
-        swap_count++;
-        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 deadline

Description: The `deposit` function accepts 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 unfavorable.

Impact: Transactions could be sent when market condition are unfavorable to deposit, even when adding a deadline parameter.

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

Recommended Mitigation: Consider making a 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: What the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTrans` 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. When it comes to auditing smart contracts, there are a lot of nitty-gritty details that one needs to pay attention to in order to prevent possible vulnerabilities.

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, outputReserves);
+         output = getOutputAmountBasedOnInput(inputAmount,
    inputReserves, outputReserves);

```

```diff
-         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_PoolDoesNotExist` is not used and should be removed

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

[I-2] `PoolFactory::constructor` lacking zero address check

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

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

```diff
-     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

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

[I-5] `TSwapPool` events should be indexed

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