Prepared by: mhmojtaba
Lead security researcher: mhmojtaba

- Audit Date: December 17, 2024

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.

Disclaimer

The `mhmojtaba` 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
	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 the following 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

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	9
Gas Optimizations	0
Total	16

Findings

High Risk Findings

[H-1] Protocol may take too many tokens from users during swap, resulting is lost fee in `TSwapPool::getInputAmountBasedOnOutput`.

Description: The `TSwapPool::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 10000 instead of 1000.

```
     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);
    }
```

Impact:
As a result, users swapping tokens via the swapExactOutput function will pay far more tokens than expected for their trades. This becomes particularly risky for users that provide infinite allowance to the TSwapPool contract. Moreover, note that the issue is worsened by the fact that the swapExactOutput function does not allow users to specify a maximum of input tokens, as is described in another issue in this report.

It's worth noting that the tokens paid by users are not lost, but rather can be swiftly taken by liquidity providers. Therefore, this contract could be used to trick users, have them swap their funds at unfavorable rates and finally rug pull all liquidity from the pool.

Proof of Concept: Add the following code to `TSwapPool.t.sol`. You'll see although the user has 10e18 balance in both tokens and should be able to swap the `swapAmount` in both ways `swapExactInput` or `swapExactOutput`, the swap fails as the fee is calculated incorrectly by `getInputAmountBasedOnOutput`. If you uncomment the commented lines and test `swapExactInput` function, you'll see the swap succeeds as in this case the fee is calculated correctly by `getOutputAmountBasedOnInput`.

```
function testRevertInswapExactOutput() 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 wethUserBalance = weth.balanceOf(address(user));
        uint256 poolTokenUserBalance = poolToken.balanceOf(address(user));
        uint256 swapAmount = 1e18;

        vm.startPrank(user);
```

```
        console.log("poolTokenUserBalance", poolTokenUserBalance);
        console.log("wethUserBalance", wethUserBalance);
        poolToken.approve(address(pool), 10e18);

        vm.startPrank(user);

        // pool.swapExactInput(
        //     poolToken,
        //     swapAmount,
        //     weth,
        //     1,
        //     uint64(block.timestamp)
        // );
        // uint256 wethBalanceAfter = weth.balanceOf(address(user));
        // uint256 poolTokenBalanceAfter = poolToken.balanceOf(address(user));
        // console.log("poolTokenBalanceAfter", poolTokenBalanceAfter);
        // console.log("wethBalanceAfter", wethBalanceAfter);

        vm.expectRevert();
        pool.swapExactOutput(
            poolToken,
            weth,
            swapAmount,
            uint64(block.timestamp)
        );
    }
```

Recommended Mitigation: The calculation of the inputBasedOnOutput is incorrect. The correct calculation is:

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

[H-2] Lack of Slippage Protection in `TSwapPool::swapExactOutput` causes users potentially recieve fewer tokens than expected.

Description: The `TSwapPool::swapExactOutput` function does not include any slippage protection, which could lead to users receiving fewer tokens than expected. This function is similar to the `swapExactInput` function, where users specify the `minOutputAmount` of token they expected to recieve, `swapExactOutput` should specify the `maxInputAmount` of token they are willing to pay.

Impact: If market changed before the transaction is executed, the user could end up doing a worse swap.

Proof of Concept:

1. The price of 1 weth is 100 usdc now.
2. User inputs `swapExactOutput` looking for 1 weth.
    1. inputToken = weth
    2. outputToken = usdc
    3. outputAmount = 100 usdc
    4. deadline = block.timestamp
3. The function does not offer a maxInput amount.
4. while transaction is pending, the price of 1 weth is 300 usdc.
5. User paid 300 usdc and recieve 1 weth instead of 100 usdc.

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
        );
+       if (inputAmount > maxInputAmount) {
+       revert();
+       }
```

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

Description: The `sellPoolTokens` function is used to sell pool tokens for the weth token. User set `poolTokenAmount` to the function willing to sell and get weth but the function miscalculates the amount of weth to be recieved due to wrong function

Impact: User will swap the wrong amount of pool tokens for weth.

Proof of Concept: The `swapExactOutput` function is used to swap tokens when the exact output is known. The `swapExactInput` function is used to swap tokens when the exact input is known. In the `sellPoolTokens`, the input is known as `poolTokenAmount`, so the function should use `swapExactInput` instead of `swapExactOutput`.

Recommended Mitigation:
Changinf the implementation of the function to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept `minOutputAmount` as a parameter.

```
function sellPoolTokens(
        uint256 poolTokenAmount,
+       uint256 minOutputAmount
    ) external returns (uint256 wethAmount) {
        return
-            swapExactOutput( i_poolToken,
i_wethToken,poolTokenAmount,uint64(block.timestamp));
+            swapExactInput( i_poolToken, poolTokenAmount, i_wethToken,
minOutputAmount,uint64(block.timestamp));
    }
```

[H-4] In `TSwapPool::_swap` function, the extra tokens given to users after every 10 swaps and that BREAKS the protocol invariant of `x * y = k`.

Description: The protocol follows a strict invariant of `x * y = k`.

- `x` is the balance of poolToken
- `y` is the balance of weth
- `k` is the constant product of the two balances
  This means, whenever the balances changed, the protocol must ensure that `x * y = k` which means the ratio of the two balances must remain the same. However, the protocol does not ensure this invariant is maintained as every 10 swap transactions, the protocol gives the user extra tokens. This causes the protocol invariant to be broken. Meaning that over time, the proticol funds will be drained.

The following of the code is 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 exploit this vulnerability to drain the protocol of its funds. By doing lots of swaps, the protocol will give the user more and more tokens and the protocol will be drained of its funds.

Proof of Concept: The followwing is a sequence of transactions that will drain the protocol of its funds. This test function shows that the protocol invariant is broken after 10 swaps.

1. User make swap for 10 times and the protocol gives the user `1_000_000_000_000_000_000` extra tokens.
2. User continues to swap much more until the protocol is drained all of its funds.

▶ POC

```
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: Remmove the extra incentive mechanism. Or if the incentive mechanism is needed, the protcol should set aside token in the same way to do that.

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

Medium Risk Findings

[M-1] The `TSwapPool::deposit` missing `deadline` check, causing the transaction to be executed even if the deadline has passed

Description: The `TSwapPool::deposit` function does not check the `deadline` parameter, which could lead to the transaction being executed even if the deadline has passed. Due to documentation "deadline The deadline for the transaction to be completed by", this could lead to unexpected behavior.

Impact: user can set a deadline and expect to revert , but it'll go through that ruins the functionality

Proof of Concept: The `deadlinee` parameter is not used in the `deposit` function.

```
Warning (5667): Unused function parameter. Remove or comment out the variable name
to silence this warning.
   --> src/TSwapPool.sol:122:9:
    |
122 |         uint64 deadline
    |         ^^^^^^^^^^^^^^^
```

Recommended Mitigation: Consider adding a check for the `deadline` parameter to ensure that the transaction is executed only if the deadline has not passed.

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
+       revertIfDeadlinePassed(deadline)
```

```
        returns (uint256 liquidityTokensToMint)
    {
```

Low Risk Findings

[L-1] Wrong information is given in the `TSwapPool::_addLiquidityMintAndTransfer` function

Description: The `TSwapPool::_addLiquidityMintAndTransfer` function is used to add liquidity to the pool and mint liquidity tokens to the user. However, the function emit some information to the users which can be used off-chain, but the information is not accurate.

Impact: The information given to the user is not accurate, which can lead to missunderstanding of has happened in the transaction by off-chain tools.

Proof of Concept:
The following code is used to emit the information to the user:

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

The following code is event which is declared in the contract and must be emited to the user:

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

Recommended Mitigation: The information given to the user should be accurate, so the information should be changed to:

- src/TSwapPool.sol Line: 210

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

[L-2] Default value is returned in the `TSwapPool::swapExactInput` function results in incorrect value given to the user.

Description: The `TSwapPool::swapExactInput` function is used to swap tokens in the pool. However, the function should return the actual amount of token bought by caller, it returns a default value of 0, which can lead to incorrect value given to the user.

Impact: The return value of the function is alwaays 0, which can lead to incorrect value given to the user.

Proof of Concept: The following test code will show that user after a successful swap should get the actual output value but will get zero.

```
function testSwapExactInputReturnZero() 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);
        uint256 expected = 9e18;
        poolToken.approve(address(pool), 10e18);
        uint256 output = pool.swapExactInput(
            poolToken,
            10e18,
            weth,
            expected,
            uint64(block.timestamp)
        );
        vm.stopPrank();
        console.log("output: ", output);
        assertEq(output, 0);  // actual output : 9066108938801491315
    }
```

Recommended Mitigation:

```
function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        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(output, minOutputAmount);
```

```
+         }

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

Informational Findings

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is useless and should be removed

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

[I-2] TMissing zero address check

▶ 2 Found Instances

- Found in src/PoolFactory.sol Line: 42

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

- Found in src/TSwapPool.sol Line: 96

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

[I-3] Wrong `IERC20.name` usage in `PoolFactory::createPool` in the `string memory liquidityTokenSymbol`. the `symbol()` should be used instead of `name()`.

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

[I-4] events should have `indexed` fileds

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.

▶ 4 Found Instances

- Found in src/PoolFactory.sol Line: 37

```
    event PoolCreated(address tokenAddress, address poolAddress);
```

- Found in src/TSwapPool.sol Line: 52

```
    event LiquidityAdded(
```

- Found in src/TSwapPool.sol Line: 57

```
    event LiquidityRemoved(
```

- Found in src/TSwapPool.sol Line: 62

```
    event Swap(
```

[I-5] Magic numbers.

Avoid using magic numbers in the code. They make the code less readable and can cause confusion. Instead, use the constant variables.

▶ 2 Found Instances

- Found in src/TSwapPool.sol Line: 291

```
-->     euint256 inputAmountMinusFee = inputAmount * 997;

-->     uint256 denominator = (inputReserves * 1_000) + inputAmountMinusFee;
```

- Found in src/TSwapPool.sol Line: 314

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

[I-6] No need to emit constant variables in functions as they are constant and can't be changed

In the following code, the variables `MINIMUM_WETH_LIQUIDITY` is a constant and can't be changed. So, there is no need to emit them in the functions.

- src/TSwapPool.sol Line: 130

```
 revert
TSwapPool__WethDepositAmountTooLow(MINIMUM_WETH_LIQUIDITY,wethToDeposit);
```

[I-7] Unused local variable

In the `TSwapPool::deposit` function, the variable `poolTokenReserves` is declared but not used. So it can be removed.

- src/TSwapPool.sol Line: 138

```
 uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

[I-8] Follow CEI pattern in functions where there is an external call

Where the functionality is to make an external call, it is recommended to follow the CEI pattern. It means that effect should be before interaction and in this case the state should be changed before the external call.

- src/TSwapPool.sol Line: 192

```
else {
        // This will be the "initial" funding of the protocol. We are
starting from blank here!
        // We just have them send the tokens in, and we mint liquidity
tokens based on the weth
+       liquidityTokensToMint = wethToDeposit;
```

```
            _addLiquidityMintAndTransfer(
                wethToDeposit,
                maximumPoolTokensToDeposit,
                wethToDeposit
            );
-           liquidityTokensToMint = wethToDeposit;
        }
```

[I-9] Function `TSwapPool::swapExactInput` is one of the most important functions in the contract. This function should have a good documentation. A complete natspec should be used for this function. Another thing is that the function is not using internally through the contract. So, it should be marked as `external` rather than `public`.