Protocol Audit Report

Prepared by: SyncAudit

TSwap Protocol Audit Report

Prepared by: Abolaji M. Adedeji Lead Auditors: Abolaji M. Adedeji

- SyncAudit

Assisting Auditors:

- None

Table of contents

▶ Details

See table

About SyncAudit

Disclaimer

The SyncAudit team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

Audit Details

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

```
f426f57731208727addc20adb72cb7f5bf29dc03
```

Scope

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

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. It is similar to Uniswap. To understand Uniswap, please watch this video: Uniswap Explained

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	4
Gas Optimizations	0
Total	11

Findings

High

[H-1] 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.

Impact: Protocol takes more fees than expected from users.

Proof of Concept:

▶ Code

```
```javascript
function testFlawedSwapExactOutput() public {
    uint256 initialLiquidity = 100e18;
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), initialLiquidity);
```
```

```
        poolToken.approve(address(pool), initialLiquidity);

        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: 100e18,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // User has 11 pool tokens
        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 11e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);
        vm.startPrank(someUser);

        // User buys 1 WETH from the pool, paying with pool tokens
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, 1 ether,
    uint64(block.timestamp));
        // Initial liquidity was 1:1 so user should have paid ~1 pool token
        // However, the user spent much more than they should. The user
    started with 11 tokens but now has less than 1
        assert(poolToken.balanceOf(someUser) < 1 ether);
        vm.stopPrank();

        // The liquidity provider can rug all funds from the pool now,
        // including those deposited by the user.
        vm.startPrank(liquidityProvider);
        pool.withdraw(
            pool.balanceOf(liquidityProvider),
            1, // minWethToWithdraw
            1, // minPoolTokensToWithdraw
            uint64(block.timestamp)
        );

        assert(weth.balanceOf(address(pool)) == 0);
        assert(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 transaciton 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

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:

1. A user swaps `poolToken` for `weth` using `swapExactInput`
2. Then swaps same amount of `poolToken` for `weth` using `sellPoolTokens` but got different `weth` amount with price remaining the same.

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

▶ Code

```
    function testSellPoolTokensMiscalculatesSwaps() public {
        uint256 initialLiquidity = 100e18;
        uint256 tokenAmount = 1e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // User has 100 pool tokens
        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 100e18;
        uint256 someUserWethInitialBalance = weth.balanceOf(someUser);
        poolToken.mint(someUser, userInitialPoolTokenBalance);
        vm.startPrank(someUser);

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

        // User sells 1 pool token using sellPoolTokens
```

```
        pool.sellPoolTokens(tokenAmount);

        uint256 someUserWethBalance1 = weth.balanceOf(someUser);
        uint256 changeInWethAfter1stSwap = someUserWethBalance1 -
            someUserWethInitialBalance;

        // User sells 1 pool token using swapExactInput
        pool.swapExactInput(
            poolToken,
            tokenAmount,
            weth,
            0,
            uint64(block.timestamp)
        );

        uint256 someUserWethBalance2 = weth.balanceOf(someUser);
        uint256 changeInWethAfter2ndSwap = someUserWethBalance2 -
            someUserWethBalance1;

        // The Weth tokens received from the swaps should be the same but
they are different
        assert(changeInWethAfter1stSwap != changeInWethAfter2ndSwap);

        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));
    }
```

Additionally, it might be wise to add a deadline to the function, as there is currently no deadline. (MEV later)

[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 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 the extra incentive in the _swap function. Meaning that over time the protocol funds will be drained.

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.

Most simply put, the protocol'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 untill all the protocol funds are drained

Place the following into TSwapPool.t.sol.

▶ 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.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 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.

```
-        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-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 unfavorable.

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

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

Add the following to the `TSwapPool.t.sol` test suite.

▶ Code

```
  function testFlawedDepositDoesNotRespectSetDeadline() public {
      assert(weth.balanceOf(address(pool)) == 0);
      assert(poolToken.balanceOf(address(pool)) == 0);
      uint256 liquidityValue = 100e18;
      uint64 invalidDeadline = uint64(block.timestamp - 1);
      vm.startPrank(liquidityProvider);
      weth.approve(address(pool), liquidityValue);
      poolToken.approve(address(pool), liquidityValue);

      // Calling the deposit function with a deadline in the past should
fail but passes
      pool.deposit({
          wethToDeposit: liquidityValue,
          minimumLiquidityTokensToMint: 0,
          maximumPoolTokensToDeposit: liquidityValue,
          deadline: invalidDeadline
      });
      vm.stopPrank();

      assert(weth.balanceOf(address(pool)) == liquidityValue);
      assert(poolToken.balanceOf(address(pool)) == liquidityValue);
  }
```

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% == 17 tokens)
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
```

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 explict return statement.

Impact: The return value will always be 0, giving incorrect information to the caller.

Proof of Concept:

▶ Code

```
    function testFlawedSwapExactInputReturnsZero() public {
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: 100e18,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // User has 50 pool tokens
        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 50e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);
        vm.startPrank(someUser);

        // User Weth balance before swap is 0
        assert(weth.balanceOf(someUser) == 0);

        // User buys 1 WETH from the pool, paying with pool tokens
```

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

        // first swap returns 0
        uint256 firstSwapReturnedValue = pool.swapExactInput(
            poolToken,
            2 ether,
            weth,
            0.1 ether,
            uint64(block.timestamp)
        );

        uint256 newWethBalanceAfterSwap1 = weth.balanceOf(someUser);
        assert(firstSwapReturnedValue == 0);
        assert(newWethBalanceAfterSwap1 > 0);

        // second swap returns 0
        uint256 secondSwapReturnedValue = pool.swapExactInput(
            poolToken,
            1.5 ether,
            weth,
            0.1 ether,
            uint64(block.timestamp)
        );
        uint256 newWethBalanceAfterSwap2 = weth.balanceOf(someUser);

        assert(secondSwapReturnedValue == 0);
        assert(newWethBalanceAfterSwap2 - newWethBalanceAfterSwap1 > 0);

        vm.stopPrank();
    }
```

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);
```

```
    }
    }
```

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] `PoolFacotry::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