TSwap Audit Report

Version 1.0

s3bc40

April 23, 2025

TSwap Audit Report

s3bc40

April 23, 2025

Prepared by: Lead Auditors: - s3bc40

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. It is similar to Uniswap.

Disclaimer

s3bc40 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

Scope

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda
- In Scope:

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

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- Tokens:
  - Any ERC20 token

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

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 `TSwapPool::getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user should deposit given an amount of output tokens. However, the function miscalculates the fee, it scales by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Proof of Concept: Add this to the `TSwapPool.t.sol`

```
    function testGetInputAmountBasedOnOutputMiscalculateFees(
        uint256 outputAmount
    ) public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        // Arrange
        uint256 inputReserves = poolToken.balanceOf(address(pool));
        uint256 outputReserves = weth.balanceOf(address(pool));
        // We are applying a 0.03% fee
        outputAmount = bound(outputAmount, 1e18, outputReserves);
        uint256 numerator = ((inputReserves * outputAmount) * 1_000);
        uint256 denominator = ((outputReserves - outputAmount) * 997);
        vm.assume(denominator > 0); // Avoid division by zero
        uint256 expectedInputAmount = (numerator / denominator);

        // Act
        uint256 inputAmount = pool.getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

        // Assert
        assertGt(inputAmount, expectedInputAmount);
    }
```

Recommended Mitigation:

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        // 997 / 10_000 = 91.3% fees!!
        // @audit-issue HIGH wrong fees
        // IMPACT: HIGH
        // LIKELIHOOD: HIGH
        return
-            ((inputReserves * outputAmount) * 10_000) /
+            ((inputReserves * outputAmount) * 1_000) /
            ((outputReserves - outputAmount) * 997);
    }
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to receive less tokens than expected

Description: The `swapExactOutput` function does not include any form of slippage protection. This function is similar to what is done to `TSwapPool::swapExactInput`, where the function specifies a `minOuputAmount`, the `swapExactOutput` function does not specify a `maxInputAmount`.

Impact: If market conditions change, the user may receive less tokens than expected. This could lead to a loss of funds for the user.

Proof of Concept: 1. The price of WETH 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 change: 1 WETH = 10,000 USDC. 5. The transaction is executed and the user ends up paying 10,000 USDC for 1 WETH, instead of 1,000 USDC.

Recommended Mitigation:

```
function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxOutputAmount,
        uint64 deadline
    )
.
.
.

    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    inputAmount = getInputAmountBasedOnOutput(
        outputAmount,
        inputReserves,
        outputReserves
    );
+   if (inputAmout > maxInputAmount) {
+       revert();
+   }
```

[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 sell their pool tokens for the underlying assets. However, the function mismatches the input and output tokens, causing users to receive the incorrect amount of tokens.

This is due to the fact that the `swapExactOutput` function is called whereas the `swapExactInput` function should be called. The `swapExactOutput` function is intended to allow users to specify the exact amount of output tokens they want to receive, while the `swapExactInput` function allows users to specify the exact amount of input tokens they want to sell.

Impact: Users will swap the wrong tokens, which is a severe disruption to the protocol. This could lead to a loss of funds for the user.

Proof of Concept: Before running the POC, make sure to use these partially fixed functions from audit:

```
    function auditFixGetInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        // 997 / 10_000 = 91.3% fees!!
        // @audit-issue HIGH wrong fees
        // IMPACT: HIGH
        // LIKELIHOOD: HIGH
        return
            ((inputReserves * outputAmount) * 1_000) /
            ((outputReserves - outputAmount) * 997);
    }

    function auditFixedSwapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
        // uint256 maxOutputAmount, @audit-issue
        uint64 deadline
    )
        public
        revertIfZero(outputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 inputAmount)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        inputAmount = auditFixGetInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

        // @audit-issue no slippage protection
        // if (outputAmount > maxOutputAmount) {
        //     revert TSwapPool__OutputTooHigh(outputAmount, maxOutputAmount);
        // }

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

    function auditFixedSellPoolTokens(
        uint256 poolTokenAmount
    ) external returns (uint256 wethAmount) {
        // @audit-issue wrong swap it should be wethAmount not poolTokenAmount
        // or it should be a swapExactInput
        return
            auditFixedSwapExactOutput(
                i_poolToken,
                i_wethToken,
                poolTokenAmount,
                uint64(block.timestamp)
            );
    }
```

Then run this fuzz test to assess of the issue:

```
        function testSellPoolTokensReturnsTheWrongAmount(
            uint256 poolTokensAmount
        ) public {
            vm.startPrank(liquidityProvider);
            weth.approve(address(pool), 100e18);
            poolToken.approve(address(pool), 100e18);
            pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
            vm.stopPrank();

            // Arrange
            uint256 maxUserPoolTokenToSwap = poolToken.balanceOf(user) - 1e18; // To handle the fees of 0.03%
            poolTokensAmount = bound(
                poolTokensAmount,
                1e18,
                maxUserPoolTokenToSwap
            );
            vm.startPrank(user); // Prank as user before approval
            poolToken.approve(address(pool), poolToken.balanceOf(user)); // Approve the pool to spend user's poolToken
            uint256 inputReserves = poolToken.balanceOf(address(pool));
            uint256 outputReserves = weth.balanceOf(address(pool));
            uint256 expectedOutputAmount = pool.getOutputAmountBasedOnInput(
                poolTokensAmount,
                inputReserves,
                outputReserves
            );

            // Act
            uint256 outputAmount = pool.auditFixedSellPoolTokens(poolTokensAmount);
            vm.stopPrank();

            // Assert
            assert(outputAmount != expectedOutputAmount);
        }
```

Recommended Mitigation:

Consider changing the implementation to user `swapExactInput` instead of `swapExactOutput`. Note 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) {
        // @audit-issue wrong swap it should be wethAmount not poolTokenAmount
        // or it should be a swapExactInput
        return
-           swapExactOutput(
-               i_poolToken,
-               i_wethToken,
-               poolTokenAmount,
-               uint64(block.timestamp)
-           );
+           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 struct invariant of `x * y = k`, where: - `x` is the amount of pool token - `y` is the amount of WETH - `k` is a constant value that represents the product of the two token amounts in the pool.

This means, that whenever the balance change in the protocol, the ratio of the two tokens 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 of 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 funds by doing a lot of swaps and collecting extra incentive given out by the protocol.

Most simply put, the protocol 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 in `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();

        // Arrange
        uint256 outputWeth = 1e17;
        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);
        poolToken.mint(user, 1e18);

        // Act
        // Swap 10 times to trigger the invariant
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        for (uint i = 0; i < 10; i++) {
            pool.swapExactOutput(
                poolToken,
                weth,
                outputWeth,
                uint64(block.timestamp)
            );
        }
        vm.stopPrank();

        // Assert
        // Check if the invariant is broken
        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++;
-    if (swap_count >= SWAP_COUNT_MAX) {
-        swap_count = 0;
-        outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
-    }
```

Medium

[M-1] `TSwapPoll::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 "The deadline for the transaction to be completed by". However, the parameter is not used in the function. This means that the transaction can be completed even after the deadline has passed, which could lead to unexpected behavior.So a user can add liquidity to the pool might be executed at unexpected times, in market conditions that are not favorable to the user.

Impact: Transactions could be sent when the market conditions are not favorable to the deposit, even if the `deadline` is set.

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,
        // @audit-issue HIGH param is not used but potentially crucial
        // if someone sets a deadline next block, they could still
           deposit
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
```

Low

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

Description: When `LiquidityAdded` event emitted in the `TSwapPool::_addLiquidityMintAndTransfer` function, it logs the parameters out of order. The `poolTokenToDeposit` value should be in the third parameter position.

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 `ouput`, it is never assigned a value, nor uses an explicit return statement.

Impact: The return value will always be 0.

Proof of Concept: Add this function the `TSwapPool.t.sol`

```
function testSwapExactInputAlwaysReturnZero(
    uint256 inputAmount,
    uint256 minOutputAmount
) public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    // Arrange
    // Bound max to the user balance for allowance
    inputAmount = bound(inputAmount, 1e18, poolToken.balanceOf(user));
    minOutputAmount = bound(minOutputAmount, 1e18, weth.balanceOf(user)
       );
    // Check if we do not trigger TSwapPool__OutputTooLow error
    uint256 inputReserves = poolToken.balanceOf(address(pool));
    uint256 outputReserves = weth.balanceOf(address(pool));
    uint256 outputAmount = pool.getOutputAmountBasedOnInput(
        inputAmount,
        inputReserves,
        outputReserves
    );
    vm.assume(outputAmount > minOutputAmount); // Avoid triggering the
       revert

    // Act
    // Pranking user to swap exact input
    vm.startPrank(user);
    poolToken.approve(address(pool), 100e18);
    uint256 outputReturned = pool.swapExactInput(
        poolToken,
        inputAmount,
        weth,
        minOutputAmount,
        uint64(block.timestamp)
    );
    vm.stopPrank();

    // Assert
    assertEq(outputReturned, 0);
}
```

Recommended Mitigation:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    returns (
        // @audit-issue LOW wrong return
-        uint256 output
+        uint256 outputAmount
    )
{
```

Informational

[I-1] `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()
+     IERC20(tokenAddress).symbol()
);
```