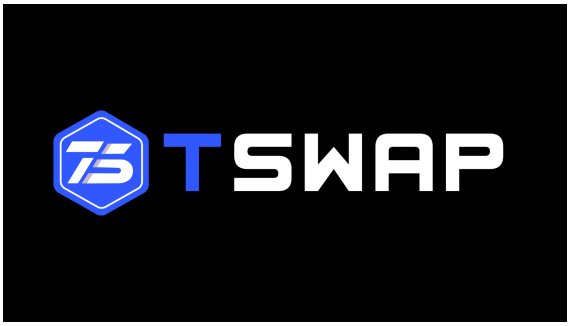

T-Swap Audit Report

Version 1.0

Cyfrin.io

May 23, 2024

Protocol Audit Report

Seven Cedars

May 23, 2024

Prepared by: Seven Cedars Lead Auditors: Seven Cedars

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 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

Scope

```
./src/
#-- PoolFactory
#-- 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

Severity	Number of Issues found
high	4
medium	2
low	2
info	11
total	19

Issues found

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protol 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 tokens of output tokens. However, the function currently miscalculates the amount. It scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Recommended Mitigation:

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

[H-2] No Slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens.

Description: The `swapExactOutput` does not include any sort of slippage protection. This function is similar to `TSwapPool::swapExactInput` where the function specifies `minOutputAMount`. The `swapExactOutput` function should include a `maxInputAMount`.

Impact: If market conditions change, user might pay far more for tokens than they expected.

Proof of Concept: 1. Price of 100 weth is now 100 poolToken. 2. User inputs `swapExactOutput` looking for 10 weth. 1. inputToken = poolToken 2. output token = weth 3. output amount = 10 4. deadline is blocknumber. 3. Funciton does not offer maxInputAmount. 4. While transaction is in mempool. Someone swaps pooltoken for weth. - the exact same trade. 5. poolToken will now drop in value, meaning that user will pay more. If this happens with Huge amounts, the difference will be huge.

PoC - step 1: fix finding [H-1] Incorrect fee calculation, as this render proper exchange based on output incorrect. In `TSwapPool::getInputAmountBasedOnOutput` change the following lines:

```
1  {
2  - return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
       outputAmount) * 997);
3  + return ((inputReserves * outputAmount) * 1_000) / ((outputReserves -
       outputAmount) * 997);
4    }
```

Step 2: Place the dollowing in `TSwapPool.t.sol`:

```
1      function test_lackingSlippageProtection () 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          // After we swap, there will be ~110 tokenA, and ~91 WETH
9          // 100 * 100 = 10,000
10         // 110 * ~91 = 10,000
11         // meaning we should expect to pay ~11 in tokenA for ~9 in weth
               .
12         uint256 expectedPayment = 11e18;
13
14         // The liquidity provider now comes around and does a big swap
               poolTokens -> wEth
15         uint256 minOutputAmount = 9e18;
16         vm.startPrank(liquidityProvider);
17             poolToken.approve(address(pool), 100e18);
18             pool.swapExactInput(
```

```
                poolToken, // = inputToken
                25e18, // = inputAMount
                weth, // = outputToken
                minOutputAmount, // = outputAmount
                uint64(block.timestamp)); // deadline
        vm.stopPrank();

        uint256 poolTokenBalanceuserBefore = poolToken.balanceOf(user);

        vm.startPrank(user);
        poolToken.approve(address(pool), 100e18);
        pool.swapExactOutput(
            poolToken, // = inputToken
            weth, // = outputToken
            9e18, // = outputAmount
            uint64(block.timestamp)); // deadline
        vm.stopPrank();

        uint256 poolTokenBalanceUserAfter = poolToken.balanceOf(user);
        console.log("Difference between expected and actual payment: ",
            (poolTokenBalanceuserBefore - poolTokenBalanceUserAfter) -
           expectedPayment);

        assert(poolTokenBalanceuserBefore - poolTokenBalanceUserAfter >
            expectedPayment);
    }
```

Recommended Mitigation: We should include a `maxInputAmount` so the user has a guarantee they will only spend up until a certain amount.

```
    function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxInputAmount,
        uint64 deadline
    )
.
.
.
        inputAmount = getInputAmountBasedOnOutput(outputAmount,
           inputReserves, outputReserves);
        _swap(inputToken, inputAmount, outputToken, outputAmount);
+ if (inputamount > maxInputAmount) {
+  revert();
+ }
```

[H-3] `TSwapPool::sellPoolTokens` mismatches input an doutput tokens, causing users to receive the incorrect amount of tokens.

Description: The `sellPoolTokens` is intended to allow users to easily sell poolTokens for weth in exchange. Users indicate how many poolTokens they are willing to sell. How the function currently miscalculates the swapped amount.

This is because the `swapExactOutput` is called, instead of the `swapExactInput`.

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

Proof of Concept: 1. User wants to exchange 10 poolTokens. 2. User inputs the `sellPoolTokens` function: 1. poolTokenAmount = 10 3. Function calls `swapExactOutput` 1. poolToken = token to retrieve from user 2. wethToken = token to send to user 3. ouputAmount = 10 (= amount of weth to send to user). 4. uint64(block.timestamp) 4. The amount of poolTokens is calculated on the basis of costing 10 weth, instead of sending 10 poolTokens for variable amount of weth. 5. Resulting in incorrect swap.

PoC - step 1: fix finding [H-1] Incorrect fee calculation, as this render proper exchange based on output incorrect. In `TSwapPool::getInputAmountBasedOnOutput` change the following lines:

```
1  {
2  - return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
       outputAmount) * 997);
3  + return ((inputReserves * outputAmount) * 1_000) / ((outputReserves -
       outputAmount) * 997);
4    }
```

Place the dollowing in `TSwapPool.t.sol`:

```
1     function test_incorrectAmountsAtSellPoolTokens () 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          uint256 poolTokensToSell = 5e18;
9          uint256 poolTokenBalanceUserBefore = poolToken.balanceOf(user);
10
11         vm.startPrank(user);
12         poolToken.approve(address(pool), 100e18);
13         weth.approve(address(pool), 100e18);
14         pool.sellPoolTokens(poolTokensToSell);
15         vm.stopPrank();
16
17         uint256 poolTokenBalanceUserAfter = poolToken.balanceOf(user);
```

```
18
19          assert(poolTokenBalanceuserBefore - poolTokenBalanceUserAfter
                != poolTokensToSell);
20      }
```

Recommended Mitigation: Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` functionality to accept a new parameter: `minWethToReceive` as it needs to be passed to `swapExactOutput`.

```
1       function sellPoolTokens(uint256 poolTokenAmount) external returns (
            uint256 wethAmount) {
2   -       return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount
        , uint64(block.timestamp));
3   +       return swapExactInput(i_poolToken, poolTokenAmount,  i_wethToken
        , minWethToReceive, uint64(block.timestamp));
4       }
5
6   Additionally, it would be good to add a deadline to the function as
        there currently is none.
```

[H-4] In `TSwapPool::_swap` the extra tokens goven to users after every `swapCount`, breaks the 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 the WETH - `k` the contant product of the two balances.

This means that whenever the balance change in the protocol, the ration between the two amounts should remain constant. However, this is broken due to the extra incentive in the `_swap` function. It means that the funds will be drained over time.

The following block of code is responsible for the issue:

```
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           }
```

Impact: A user can drain the protocol by making many swaps.

In short, the protocols core invariant is broken.

Proof of Concept: 1. The user swaps 10 times and collects the extra incentive tokens. 2. The user continues to swap until all the protocols funds are drained.

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;
        uint256 numberOfSwaps = 10;
        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);

        vm.startPrank(user);
        for (uint256 i; i < numberOfSwaps; i++) {
            poolToken.approve(address(pool), type(uint256).max);
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
        }
        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);

        vm.assertEq(expectedDeltaY, actualDeltaY);
    }
```

Recommended Mitigation: Remove the extra incentive mechanism is the most straightforward solution.

```
-     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] At `TSwapPool::deposit` the `deadline` parameter is set but not used. As a result, a user deposit that should fail, will pass. Severe disruption of protocol.

Description: At the `TSwapPool::deposit` the `deadline` parameter is ignored. It means the function allows deposits even though the deadline has passed.

Impact: Transactions can be completed at moments after deadline, possibly during averse market

conditions.

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

Recommended Mitigation: Include a require check to check if the deadline passed. The checks can be used in the form of already existing modifiers:

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

[M-2] Rebase, fee-on-transfer and ERV-777 tokens break the protocol.

Any token that adds functionality to the basic transfer functionality of ERC-20 will cause the invariant of the protocol to break.

Low

[L-1] The `TSwapPoll::LiquidityAdded` event has parameters out of order.

Description: When the `LiquidityAdded` event is emitted at the `TSwapPoll::_addLiquidityMintAndTran` function, the `poolTokensToDeposit` and `wethToDeposit` are swapped.

Impact: Event emits incorrect information, leading to off-chain functions potentially malfunctioning.

Recommended Mitigation:

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

[L-2]: At function `TSwapPool::swapExactInput` the `uint256 output` returns incorrect value .

Description: The `swapExactInput` function is expected to return the actual amount of token bought by the user. However, while it has a return parameter `output`, it is never assigned a value. It also does not use an explicit return statement.

Impact: The return value will always be 0.

Proof of Concept:

Recommended Mitigation:

- Found in src/TSwapPool.sol

```
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (
-           uint256 output
+           uint256 outputAmount
        )
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
            inputReserves, outputReserves);

        if (outputAmount < minOutputAmount) {
            revert TSwapPool__OutputTooLow(outputAmount,
               minOutputAmount);
        }

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

Informational

[I-1]: `public` functions not used internally should be marked `external`, to save gas.

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

- Found in src/TSwapPool.sol:

```
    function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
// The following line should be external.
public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 output)
```

[I-2]: Avoid use of magic numbers: Define and use `constant` variables instead of using literals.

Using `constant` variable increases readability of code and decreases chances of inadvertantly introducing errors.

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

- Found in src/TSwapPool.sol:

```
        uint256 inputAmountMinusFee = inputAmount * 997;
```

- Found in src/TSwapPool.sol:

```
        return ((inputReserves * outputAmount) * 10000) / ((
            outputReserves - outputAmount) * 997);
```

- Found in src/TSwapPool.sol:

```
        1e18, i_wethToken.balanceOf(address(this)), i_poolToken.
            balanceOf(address(this))
```

- Found in src/TSwapPool.sol:

```
        1e18, i_poolToken.balanceOf(address(this)), i_wethToken.
            balanceOf(address(this))
```

- Found in src/TSwapPool.sol:

```
          uint256 denominator = (inputReserves * 1000) +
              inputAmountMinusFee;
```

[I-3]: Large literal values multiples of 10000 can be replaced with scientific notation. Using scientific notation increases readability of code and decreases chances of inadvertantly introducing errors.

Use `e` notation, for example: `1e18`, instead of its full numeric value.

- Found in src/TSwapPool.sol: `javascript uint256 private constant MINIMUM_WETH_LIQUIDIT = 1_000_000_000;`
- Found in src/TSwapPool.sol: `javascript return ((inputReserves * outputAmount )* 10000)/ ((outputReserves - outputAmount)* 997);`
- Found in src/TSwapPool.sol: `javascript outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);`

[I-4]: Unused Custom Error, remove to save gas.

It is recommended that the definition be removed when custom error is unused.

- Found in src/PoolFactory.sol:

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

[I-5]: Absent address(0) check, include to avoid uncaught errors.

- Found in src/PoolFactory.sol:

```
1       constructor(address wethToken) {
2         i_wethToken = wethToken;
3     }
```

[I-6]: 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/PoolFactory.sol

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

- Found in src/TSwapPool.sol

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

- Found in src/TSwapPool.sol

```
1      event LiquidityRemoved(address indexed liquidityProvider,
          uint256 wethWithdrawn, uint256 poolTokensWithdrawn);
```

- Found in src/TSwapPool.sol

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

[I-7]: At `TSwapPool::deposit` the `poolTokenReserves` parameter is unused and can be removed to save gas.

```
1    int256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

[I-8]: At `TSwapPool::deposit` it is better to set `liquidityTokensToMint` before `_addLiquidityMintAndTransfer` is called to follow CEI conventions.

- Found in src/TSwapPool.sol

```
1  +     liquidityTokensToMint = wethToDeposit;
2       _addLiquidityMintAndTransfer(wethToDeposit,
           maximumPoolTokensToDeposit, wethToDeposit);
3  -     liquidityTokensToMint = wethToDeposit;
```

[I-9]: Function `TSwapPool::swapExactInput` misses a natspec, please include to increase readbility of code and decrease chances of introrducing bugs.

Natspecs are meant to increase understanding of code for, both internal and external, developers working in the code base. Lacking natspecs increases the chance for introducing bugs due to poor understanding of code.

- Found in src/TSwapPool.sol

```
1     function swapExactInput(
```

[I-10]: Function `TSwapPool::totalLiquidityTokenSupply` should be set to external, to save gas.

Any function only called externally, can be set to external to save gas.

- Found in src/TSwapPool.sol

```
    function totalLiquidityTokenSupply() public view returns (uint256)
        {
         return totalSupply();
    }
```

[I-11]: Function `PoolFactory::createPool`, the `liquidityTokenSymbol` should read from `.symbol()` not `.name()`.

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