TSwap Protocol Audit Report

Version 1.0

Lance Addison

December 17, 2024

TSwap Protocol Audit Report

Lance Addison

December 17, 2024

Prepared by: Cyfrin Lead Auditors: - Lance Addison

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - Highs
    * [H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes the protocol to take too many tokens from users, resulting in higher fees than expected
    * [H-2] Lack or slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens
    * [H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens

    * [H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`
- Mediums
    * [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline
    * [M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol invariant
- Lows
    * [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::PoolFactory__PoolDoesNotExist` is not used and should be removed
    * [I-2] Lacking zero address checks
    * [I-3] `PoolFactory::createPool liquidityTokenSymbol` should use `.symbol()` instead of `.name()`
    * [I-3] Event is missing `indexed` fields
    * [I-4] `MINIMUM_WETH_LIQUIDITY` in `TSwapPool::deposit` is a constant and therefore doesn't need to be emitted
    * [I-5] In `TSwapPool::deposit` it is better to set `liquidityTokensToMint` before calling the `_addLiquidityMintAndTransfer` function to follow the CEI pattern
    * [I-6] Use of "magic" numbers is discouraged
    * [I-7] `public` functions not used internally could be marked `external`
    * [I-8] The natspec is missing for `TSwapPool::swapExactInput` and should be written to avoid confusion when calling the function
- Gas
    * [G-1] In `TSwapPool::deposit` the `poolTokenReserves` variable is unused and should be removed

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

Disclaimer

Lance Addison 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

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda
- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- Tokens:
  - Any ERC20 token

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

I had a great time auditing this protocol! Patrick did a great job teaching this section and taught me about many new vulnerabilities.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	8
Gas	1
Total	17

Findings

Highs

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes the protocol to take too many tokens from users, resulting in higher fees than expected

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of input 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 han expected from users.

Proof of Concept:

1. The user calls 'TSwapPool::swapExactOutput
2. The incorrect `inputAmount` is calculated from the `getInputAmountBasedOnOutput` function and the users pays too much in fees

Proof Of Code

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

```
    function testFeesTooHigh() public {
        uint256 outputWethAmount = 10 ether;
        uint256 poolTokenReserves = 100 ether;
        uint256 wethReserves = 100 ether;

        vm.prank(user);
        uint256 actualPoolTokens = pool.getInputAmountBasedOnOutput(
            outputWethAmount, poolTokenReserves, wethReserves);

        uint256 expectedPooltokens = ((poolTokenReserves *
            outputWethAmount) * 1000) / ((wethReserves -
            outputWethAmount) * 997);

        assertEq(expectedPooltokens, actualPoolTokens);
    }
```

Recommended Mitigation:

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
```

```
            public
            pure
            revertIfZero(outputAmount)
            revertIfZero(outputReserves)
            returns (uint256 inputAmount)
        {
-           return ((inputReserves * outputAmount) * 10000) / ((
        outputReserves - outputAmount) * 997);
+           return ((inputReserves * outputAmount) * 1000) / ((
        outputReserves - outputAmount) * 997);
        }
```

[H-2] Lack or 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 = 1e18 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.

Proof Of Code

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

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

        // Market changes significantly!
        uint256 inputAmount = 50e18;
        uint256 minOutputAmount = 10e18;
        vm.startPrank(user2);
        weth.approve(address(pool), inputAmount);
        pool.swapExactInput(weth, inputAmount, poolToken,
            minOutputAmount, uint64(block.timestamp));
        vm.stopPrank();

        uint256 outputAmount = 1e18;
        vm.startPrank(user);
        weth.approve(address(pool), type(uint256).max);
        weth.mint(user, 100e18);
        uint256 actualInputAmount = pool.swapExactOutput(weth,
           poolToken, outputAmount, uint64(block.timestamp));
        vm.stopPrank();

        uint256 expectedInputAmount = 10131404313951956880;

        assertEq(actualInputAmount, expectedInputAmount);
    }
```

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,
        IERC20 outputToken,
        uint256 outputAmount,
+       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 user 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 want to swap the wrong amount of tokens, which is a severe disruption of protocol functionality.

Proof of Concept:

Proof Of Code

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

```
    function testSellPoolTokensWrongAmount() 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 poolTokenAmount = 9e17;

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        uint256 outputTokens = pool.sellPoolTokens(poolTokenAmount);
        // to account for the extra fees charged in the `TSwapPool::
            getInputAmountBasedOnOutput` function
        outputTokens /= 10;
        vm.stopPrank();

        uint256 expectedOutputAmount = pool.getOutputAmountBasedOnInput
            (poolTokenAmount, 100e18, 100e18);

        assertEq(outputTokens, expectedOutputAmount);
    }
```

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
    , minWethToRecieve, 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 tokens - `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 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 into `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();

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

Mediums

[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 send when market conditions are unfavorable to deposit, even when adding a deadline parameter.

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,
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
```

[M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol invariant

Description: If a token within a trading pair implements unusual behaviors like fee-on-transfer, rebase, or follows the ERC-777 standard, it can disrupt the core invariant of the Automated Market Maker (AMM) protocol where `x * y = k`. This invariant is crucial for maintaining the balance between the two tokens in a liquidity pool.

Impact: - **Liquidity Imbalance:** The unexpected changes in token supply or value due to fees or rebasing can cause the actual product of reserves to deviate from `k`, leading to an imbalance in liquidity. - **Price Slippage:** Users might experience significant price slippage, as the price calculation based on the invariant will no longer be accurate. - **Arbitrage Opportunities:** This imbalance can be exploited by arbitrageurs, potentially draining liquidity or causing further price manipulation. - **Loss of Trust:** If users or liquidity providers notice these issues, it could lead to a loss of trust in the protocol, reducing its usage and liquidity.

Proof of Concept: - **Fee-on-Transfer**: When a token transfer includes a fee, the amount of tokens received by the contract differs from the amount sent, breaking the balance expected by `x * y = k`. - Example: Token A charges a 1% fee on transfer. If 100 tokens are supposed to be added to the pool, only 99 arrive, but the system assumes 100, thus `k` is incorrect. - **Rebase Tokens**: These tokens can change their total supply after each transaction, directly affecting the `x` or `y` in the equation without corresponding changes to the other side. - Example: An elastic supply token like AMP might rebase, reducing or increasing its supply, which would require manual adjustment of `k` to maintain the invariant. - **ERC-777**: This standard allows for hooks that can modify token transfers, potentially adding or subtracting tokens in ways not anticipated by the AMM logic.

Recommended Mitigation: - **Token Validation:** Implement checks to detect and reject tokens with known problematic behaviors during pool creation: - Use token standards metadata or external registries to identify fee-on-transfer or rebasing tokens. - Scan for known problematic token addresses. - **Invariant Adjustment Mechanism:** Develop a mechanism to dynamically adjust `k` when there are known or detected supply changes due to rebases or fees: - This could involve tracking token transfers and adjusting `k` proportionally to any fees or supply changes. - **Price Impact Limiters:** Introduce safeguards or warnings when the price impact due to slippage exceeds certain thresholds, alerting users and possibly pausing transactions to recalibrate. - **Documentation and User Education:** Clearly document which types of tokens are not supported and educate users on the risks of using such tokens in the protocol to manage expectations and reduce misuse. - **Continuous Monitoring:** Regularly audit and monitor the pools for any signs of invariant deviation, with automated systems to flag anomalies for further investigation or automatic correction.

Lows

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

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran`
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, wethToDeposit, poolTokensToDeposit)
   ;
-   emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit)
   ;
```

[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.

Proof of Concept:

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

```
    function testSwapExactInputNoReturn() 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 inputAmount = 1e18;
        uint256 minOutputAmount = 1e17;
        vm.startPrank(user);
        weth.approve(address(pool), inputAmount);
        uint256 outputAmount = pool.swapExactInput(weth, inputAmount,
            poolToken, minOutputAmount, uint64(block.timestamp));
        vm.stopPrank();

        uint256 actualOutputAmount = 987158034397061298;

        assertEq(outputAmount, actualOutputAmount);
    }
```

Recommended Mitigation:

```diff
    {
        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, 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

`PoolFactory.sol`:

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

`TSwapPool.sol`:

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

[I-3] `PoolFactory::createPool` `liquidityTokenSymbol` 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-3] 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.

4 Found Instances

- Found in src/PoolFactory.sol Line: 35

  ```
      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-4] `MINIMUM_WETH_LIQUIDITY` in `TSwapPool::deposit` is a constant and therefore doesn't need to be emitted

`MINIMUM_WETH_LIQUIDITY` is a constant variable and therefore doesn't need to be emitted in the `TSwapPool::TSwapPool__WethDepositAmountTooLow` event to save gas.

[I-5] In `TSwapPool::deposit` it is better to set `liquidityTokensToMint` before calling the `_addLiquidityMintAndTransfer` function to follow the CEI pattern

It is best practice to follow the CEI pattern and make sure any interactions with external contracts are handled last.

```
.
.
.
          } else {
+             liquidityTokensToMint = wethToDeposit;

              _addLiquidityMintAndTransfer(wethToDeposit,
                  maximumPoolTokensToDeposit, wethToDeposit);

-             liquidityTokensToMint = wethToDeposit;
          }
```

[I-6] Use of “magic” numbers is discouraged

It can be confusing to see number literals in a codebase, and it’s much more readable if the numbers are given a name.

It is recommended to change the number literals in the `TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool::getInputAmountBasedOnOutput` functions.

Example `TSwapPool::getOutputAmountBasedOnInput`:

```
        uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * 1000) +
            inputAmountMinusFee;
```

Instead, you could use:

```
uint256 public constant FEE_PERCENTAGE = 997;
uint256 public constant FEE_PRECISION = 1000;
.
.
.
function getOutputAmountBasedOnInput
.
.
.
    uint256 inputAmountMinusFee = inputAmount * FEE_PERCENTAGE;
    uint256 numerator = inputAmountMinusFee * outputReserves;
    uint256 denominator = (inputReserves * FEE_PRECISION) +
        inputAmountMinusFee;
    return numerator / denominator;
}
```

[I-7] `public` functions not used internally could be marked `external`

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

1 Found Instances

- Found in src/TSwapPool.sol Line: 298

  ```
      function swapExactInput(
  ```

[I-8] The natspec is missing for `TSwapPool::swapExactInput` and should be written to avoid confusion when calling the function

Gas

[G-1] In `TSwapPool::deposit` the `poolTokenReserves` variable is unused and should be removed

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