TSwap Protocol Audit Report

Version 1.0

Chainrunner

February 11, 2024

TSwap Protocol Audit Report

Chainrunner

February 11, 2024

Prepared by: Chainrunner Lead Auditors:

- Dibakar Sutra Dhar

Table of Contents

Protocol Summary

TSWAP is an constant-product AMM that allows users permissionlessly trade WETH and any other ERC20 token set during deployment. Users can trade without restrictions, just paying a tiny fee in each swapping operation. Fees are earned by liquidity providers, who can deposit and withdraw liquidity at any time.

Disclaimer

The Chainrunner 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

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda
- In 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

Severtity	Number of issues found
High	4
Medium	2
Low	2
Info	11
Gas	1
Total	20

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 intented to calculate the amount of tokens an user should deposit given an amount of tokens of output tokens. However, the function currently miscalculates the resulting amount. When calculating the fees, it scales the amount by `10_000` instead of `1_000`.

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: To test this, include the following code in the `TSwapPool.t.sol` file:

```
function testFlawedSwapExactOutput() public {
    uint256 initialLiquidity = 100e18;
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), initialLiquidity);
    poolToken.approve(address(pool), initialLiquidity);
    pool.deposit(initialLiquidity, 0, initialLiquidity, uint64(block.
        timestamp));
    vm.stopPrank();

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

    console.log("Start: ", poolToken.balanceOf(someUser));

    vm.startPrank(someUser);
    // someUser buy 1 WETH from the pool, paying with poolTokens
    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, it spent much more than that. The user started with 11
        tokens, and now only has less than 1.
    console.log("End: ", poolToken.balanceOf(someUser));
    assertLt(poolToken.balanceOf(someUser), 1 ether);
    vm.stopPrank();

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

    assertEq(weth.balanceOf(address(pool)), 0);
    assertEq(poolToken.balanceOf(address(pool)), 0);
    vm.stopPrank();
}
```

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 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 similiar to what is done in `TSwapPool::swapExactInput`, where the function specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact: If market condition changes before the transaction processes, the user could get a much worse swap.

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 swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
+   uint256 maxInputAmount
)
.
.
.
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    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 recieve the incorrect amount of tokens

Description: The `sellPoolTokens` function is intented to allow users to easily sell pool tokens and recieve WETH in exchange. Users indicate how many poolToken they are 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 protocl functionality.

Proof of Concept:

```
function test_sellPoolTokensMiscalculations() public {
    uint256 initialLiquidity = 100e18;
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), initialLiquidity);
    poolToken.approve(address(pool), initialLiquidity);
    pool.deposit(initialLiquidity, 0, initialLiquidity, uint64(block.
       timestamp));
    vm.stopPrank();

    uint256 sellAmount = 5e18;

    uint256 inputReserves = poolToken.balanceOf(address(pool));
    uint256 outputReserves = weth.balanceOf(address(pool));

    uint256 expectedOutputAmount = pool.getOutputAmountBasedOnInput(
       sellAmount, inputReserves, outputReserves);

    vm.startPrank(user);
    poolToken.mint(user, 15e18);
    poolToken.approve(address(pool), type(uint256).max);
    uint256 actualOutputAmount = pool.sellPoolTokens(sellAmount);
    vm.stopPrank();

    assertEq(expectedOutputAmount, actualOutputAmount);
}
```

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 `minWethToRecieve` to be passed to `swapExactInput`).

```
function sellPoolTokens(
    uint256 poolTokenAmount
+   uint256 minWethToRecieve
) 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));
}
```

Additionaly, it might be wise to add a deadline to the function, as there is currently no deadnline. Possible MEV attack.

[H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swap_count` breaks the protocol invariant of `x * y = k`

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

- `x`: The balance of 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 ration 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 protocl fund 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 of the protocol.

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.

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

```
function test_InvariantBroken() 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.mint(user, outputWeth * 10);
    poolToken.approve(address(pool), type(uint256).max);
    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();

    int256 endingY = int256(weth.balanceOf(address(pool)));
    int256 actualDeltaY = endingY - 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)
    ;
-    }
```

Medium

[M-1] `TSwapPool::deposit` is missing `deadline` check causing transactions to complete even after the deadline

Description: The `deposit` function accepts a `deadline` paramenter, 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 rates, 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 changes 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 ERC777 tokens breaks protocol invariant

Description: The core invariant of the protocol is: `x * y = k`. In practice though, the protocol takes fees and actually increases k. So we need to make sure `x * y = k` before fees are applied.

Low

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

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran` function, it logs value 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: Swap the parameter position of `poolTokensToDeposit` and `wethToDeposit` in the event emitter -

```diff
-    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 with 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: Add the following code to the `TSwapPool.t.sol` -

```
    function test_swapExactInputReturnsZero() public {
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit(initialLiquidity, 0, initialLiquidity, uint64(
            block.timestamp));
        vm.stopPrank();

        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 11e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);

        vm.startPrank(someUser);
        // user sell 10 poolTokens to the pool, recieves some over 9
           WETH
        poolToken.approve(address(pool), type(uint256).max);
        uint256 output = pool.swapExactInput(poolToken, 10 ether, weth,
             1, uint64(block.timestamp));
        vm.stopPrank();

        assertEq(output, 0);
    }
```

Recommended Mitigation:

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

-   uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
   inputReserves, outputReserves);
+   uint256 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, ouput);
}
```

Informationals

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed

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

[I-2] `PoolFactory::constructor` lacks zero address check on `address wethToken` input parameter

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

[I-4] `TSwapPool::constructor` lacks zero address check

```
    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-5] Event is missing `indexed` fields

Description: 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 Line: 35
- Found in src/TSwapPool.sol Line: 43
- Found in src/TSwapPool.sol Line: 44

- Found in src/TSwapPool.sol Line: 45

[I-6] `TSwapPool::deposit` is emitting constant variable during revert

description: `revert TSwapPool__WethDepositAmountTooLow()` is emitting `MINIMUM_WETH_LIQUIDIT` in the error message which is a constant variable and therefore not required to be emitted.

```
-   error TSwapPool__WethDepositAmountTooLow(uint256 minimumWethDeposit
    , uint256 wethToDeposit);
+   error TSwapPool__WethDepositAmountTooLow(uint256 wethToDeposit);

    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
-           revert TSwapPool__WethDepositAmountTooLow(
   MINIMUM_WETH_LIQUIDITY, wethToDeposit);
+           revert TSwapPool__WethDepositAmountTooLow(wethToDeposit);
        }
```

[I-7] `TSwapPool::deposit` doesn't follow CEI

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

[I-8] Constants should be defined and used instead of literals

- Found in src/TSwapPool.sol Line: 229

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

- Found in src/TSwapPool.sol Line: 231

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

- Found in src/TSwapPool.sol Line: 246

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

- Found in src/TSwapPool.sol Line: 329

```
outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
```

- Found in src/TSwapPool.sol Line: 372

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

- Found in src/TSwapPool.sol Line: 378

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

[I-9] `TSwapPool::swapExactInput` has no NATSPEC documentation

[I-10] Functions not used internally could be marked external

- Found in src/TSwapPool.sol Line: 249

```
    function swapExactInput(
```

[I-11] View Functions returning contract data could be marked external

- Found in src/TSwapPool.sol Line: 434

```
    function totalLiquidityTokenSupply()
-   public
+   external
```

Gas

[G-1] Unused local variable in `TSwapPool::deposit()`

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
            revert TSwapPool__WethDepositAmountTooLow(
                MINIMUM_WETH_LIQUIDITY, wethToDeposit);
        }
        if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this))
                ;
-               uint256 poolTokenReserves = i_poolToken.balanceOf(address(
    this));
```