TSwap Initial Audit Report

Version 0.1

Cyfrin.io

October 28, 2023

TSwap Audit Report

YOUR_NAME_HERE

September 1, 2023

TSwap Audit Report

Prepared by: YOUR_NAME_HERE Lead Auditors:

- YOUR_NAME_HERE

Assisting Auditors:

- None

Table of contents

See table

    - Issues found

- Findings
    - High
        - [H-1] The `sellPoolTokens` function miscalculates amount of tokens bought
        - [H-2] Protocol may take too many tokens from users during swap, resulting is lost fee
    - Medium
        - [M-1] Rebase, fee-on-transfer, ERC777, and centralized ERC20s can break core invariant
        - [M-2] Missing deadline check when adding liquidity
        - [M-3] Lack of slippage protection in `swapExactOutput` function
    - Low
        - [L-1] Wrong values logged in `LiquidityAdded` event
        - [L-2] Swapping function returns default value
    - Informational
        - [I-1] Poor test coverage

About YOUR_NAME_HERE

Disclaimer

The YOUR_NAME_HERE 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

		Impact		
	Low	M	M/L	L

Audit Details

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

```
1ec3c30253423eb4199827f59cf564cc575b46db
```

Scope

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

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.

Roles

- Liquidity Provider: An account who deposits assets into the pool to earn trading fees.
- User: An account who swaps tokens.

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	3
Low	2
Info	0
Gas	0
Total	5

Findings

High

[H-1] The `sellPoolTokens` function miscalculates amount of tokens bought

The `sellPoolTokens` is intended to allow users easily sell pool tokens and receive WETH in exchange. Users indicate how many pool tokens they're willing to sell using 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` is the one that should be called. Because users specify the exact amount of input tokens - not output tokens.

Consider changing the implementation to use the `swapExactInput` function. Note that this would also require to change the `sellPoolTokens` function to accept a new parameter (e.g., `minWethToReceive`) to be passed down to `swapExactInput`.

```
    function sellPoolTokens(
        uint256 poolTokenAmount
+       uint256 minWethToReceive
    ) external returns (uint256 wethAmount) {
-       return swapExactOutput(
+       return swapExactInput(
            i_poolToken,
            poolTokenAmount,
            WETH_TOKEN,
+           minWethToReceive,
            uint64(block.timestamp)
        );
    }
```

[H-2] Protocol may take too many tokens from users during swap, resulting is lost fee

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 10000 instead of 1000.

```
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);
    }
```

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.

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({
        wethToDeposit: initialLiquidity,
        minimumLiquidityTokensToMint: 0,
        maximumPoolTokensToDeposit: initialLiquidity,
        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);

    // Users 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, it spent much more than that. The user started with 11 tokens, and now only has less than 1.
    assertLt(poolToken.balanceOf(someUser), 1 ether);
    vm.stopPrank();

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

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

Medium

[M-1] Rebase, fee-on-transfer, ERC777, and centralized ERC20s can break core 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.

Impact:

Proof of Concept:

Recommended Mitigation:

[M-2] Missing deadline check when adding liquidity

The `deposit` function accepts a `deadline` parameter, which according to 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 for the caller.

Consider making the following change to the `deposit` function:

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

[M-3] Lack of slippage protection in `swapExactOutput` function

The `swapExactOutput` function does not include any sort of slippage protection to protect user funds that swap tokens in the pool. Similar to what is done in the `swapExactInput` function, it should include a parameter (e.g., `maxInputAmount`) that allows callers to specify the maximum amount of tokens they're willing to pay in their trades.

```
function swapExactOutput(
    IERC20 inputToken,
+   uint256 maxInputAmount
    IERC20 outputToken,
    uint256 outputAmount,
    uint64 deadline
)
    public
    revertIfZero(outputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 inputAmount)
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    inputAmount = getInputAmountBasedOnOutput(outputAmount, inputReserves, outputReserves);

+   if (inputAmount > maxInputAmount) {
+       revert TSwapPool__OutputTooHigh(inputAmount, maxInputAmount);
+   }

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

Low

[L-1] Wrong values logged in `LiquidityAdded` event

When the `LiquidityAdded` event is emitted in the `_addLiquidityMintAndTransfer` function, it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the third place, whereas the `wethToDeposit` value should go second.

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

[L-2] Swapping function returns default value

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 never assigns a value to it, nor uses an explicit **`return`** statement.

As a result, the function will always return zero. Consider modifying the function so that it always return the correct amount of tokens bought by the caller.

Informational

[I-1] Poor test coverage

```
Running tests...
| File                 | % Lines          | % Statements     | % Branches       | % Funcs          |
| -------------------- | ---------------- | ---------------- | -------------- | -------------- |
| src/PoolFactory.sol | 100.00% (11/11) | 100.00% (16/16) | 100.00% (2/2) | 100.00% (3/3) |
| src/TSwapPool.sol   | 54.84% (34/62)  | 59.14% (55/93)  | 33.33% (6/18) | 37.50% (6/16) |
```

Recommended Mitigation: Aim to get test coverage up to over 90% for all files.