TSwap Protocol Security Review

Version 1.0

Operation-C

October 8, 2024

TSwap Protocol Security Review

Operation-C

Oct 8, 2024

Prepared by: Operation-C

Lead Security Researcher: Operation-C

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)

Disclaimer

The Operation-C 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

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.

Issues found

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

Findings

High

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

Description The `TSwapPool::getInputAmountBasedOnOutput` is inteded to calculate the amount of tokens a user should deposit given an amount of tokens 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`.

```
1       inputAmount = ((100 * 1) * 10000) / ((100 - 1) * 997);
```

Impact Protocol takes more fees than expected from users.

PoC

- Numerator: 100 * 1 * 10000 = 1,000,000. - Denominator: (100 - 1) * 997 = 99 * 997 = 98,703. - Input Amount: 1,000,000 / 98,703 = 10.13.

Thus, you need approximately 10.13 pool tokens to get 1 WETH. Since you started with 11 pool tokens, after the swap, you are left with:

```
11 - 10.13 = ~0.87 pool tokens.
```

```
    Logs:
    Starting balance of testAccount after mint: 11000000000000000000
    Ending balance of testAccount after swap:     868595686048043120
```

```
    function testInputAmount() public {
        vm.startPrank(liquidityProvider);

        // approve pool access to funds
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);

        // deposit: wethToDeposit, minLiqaidityTokensToMint,
           maxPoolTokensToDeposit, deadline
        // initial liquidity is 1:1
        pool.deposit(100e18, 0, 100e18, uint64(block.timestamp));

        vm.stopPrank();

        // user mints 11 pool tokens
        poolToken.mint(testAccount, 11e18);
        console.log("Starting balance of testAccount after mint: %s",
           poolToken.balanceOf(testAccount));

        vm.startPrank(testAccount);

        poolToken.approve(address(pool), type(uint256).max);

        // buying 1 weth, swapEactOutput makes an internal call to
           getInputAmountBasedOnOutput
        pool.swapExactOutput(poolToken, weth, 1e18, uint64(block.
           timestamp));

        assertLt(poolToken.balanceOf(testAccount), 1e18);
        vm.stopPrank();

        // assertLt(poolToken.balanceOf(testAccount), 1e18);
        console.log("Ending balance of testAccount after swap:     %s",
            poolToken.balanceOf(testAccount));
    }
```

Recommended Mitigation

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

```
7           pure
8           revertIfZero(outputAmount)
9           revertIfZero(outputReserves)
10          returns (uint256 inputAmount)
11      {
12          return
13  -           ((inputReserves * outputAmount) * 10000) / ((
        outputReserves - outputAmount) * 997);
14  +           ((inputReserves * outputAmount) * 1_000) / ((
        outputReserves - outputAmount) * 997);
15      }
```

[H-2] The `sellPoolTokens` function miscalculates amount of tokens bought

Description The `TSwapPool::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 `TSwapPool::swapExactOutput` is called, whereas the `TSwapPool::swapExactInput` is the one that should be called. Because users specify the exact amount of input tokens - not output tokens.

```
1       function sellPoolTokens(
2           uint256 poolTokenAmount
3       ) external returns (uint256 wethAmount) {
4           return
5   ->          swapExactOutput(
6                   i_poolToken,
7                   i_wethToken,
8                   poolTokenAmount,
9                   uint64(block.timestamp)
10              );
11      }
```

Impact Users are supposed to specify the exact amount of pool tokens they wish to sell, not the amount of output tokens they want to receive. The correct function to use in this scenario is `TSwapPool::swapExactInput`. Using `TSwapPool::swapExactOutput` results in a failure because the user wants to specify how many pool tokens they are selling, not how much WETH they will receive and could lead to a revert when attempting to sell their pool tokens.

Proof of Concept: 1. The user specifies they want to receive 1 WETH by selling pool tokens. 2. The contract calculates how many pool tokens are required to get 1 WETH based on the current pool reserves. 3. Let's say, based on the current reserves, the contract calculates that ~11 pool tokens are needed to obtain 1 WETH. 4. However, the user has their entire pool tokens approved for the

contract to spend. 5. When the contract tries to execute the swap, it realizes that it does not have enough allowance to perform the swap, leading to the `ERC20InsufficientAllowance` error. 6. The swap fails because the contract tries to use more tokens than it has permission for.

```
    function testSellTokens() public {
        vm.startPrank(liquidityProvider);

        // approve pool access to funds
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);

        // initial liquidity is 1:1
        pool.deposit(100e18, 0, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(testAccount);

        // minting 10 pool tokens for testAccount
        poolToken.mint(testAccount, 10e18);

        poolToken.approve(testAccount, 10e18);

        // expecting a revert since the TSwapPool::sellPoolTokens is
           calling swapExactOutput instead of swapEactInput
        vm.expectRevert();
        uint256 expectedWeth = pool.sellPoolTokens(10e18);

        vm.stopPrank();
    }
```

Recommended Mitigation Consider changing the implementation to use the `TSwapPool::swapExactInput` . Note that this would also require to change the `TSwapPool::sellPoolTokens` to accept a new parameter (e.g., `minWethToReceive`) to be passed down to `TSwapPool::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-3] 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 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 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 untill all the protocol funds are drained

```

    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));
```

```
18          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
19          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
20          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
21          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
22          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
23
24          int256 startingY = int256(weth.balanceOf(address(pool)));
25          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
26
27          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
28          vm.stopPrank();
29
30          uint256 endingY = weth.balanceOf(address(pool));
31          int256 actualDeltaY = int256(endingY) - int256(startingY);
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

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.

```
1  -        swap_count++;
2  -        // Fee-on-transfer
3  -        if (swap_count >= SWAP_COUNT_MAX) {
4  -            swap_count = 0;
5  -            outputToken.safeTransfer(msg.sender, 1
       _000_000_000_000_000_000);
6  -        }
```

Medium

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

Description The `TSwapPool::deposit` acceptes a deadline parameter, which according to the documentation is the “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 unexpexted times, in market conditions where the deposit rate is unfavorable.

Impact Transactions could be sent when market conditions are unfavorable to deposit, even when

adding a deadline parameter.

PoC The `deadline` parameter is unsed.

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
```

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] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

Description: The `TSwapPool::swapExactOutput` 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 transaciton 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 = 1 4. deadline = x value 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

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,
+       uint256 maxInputAmount,
.
.
.
        inputAmount = getInputAmountBasedOnOutput(outputAmount,
            inputReserves, outputReserves);
+       if(inputAmount > maxInputAmount){
+           revert();
+       }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
```