COMPETITIVE AUDITS

Tushar Goyal

Table of Contents

Protocol Summary

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.sol
#-- TSwapPool.sol
```

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

Issues found

Findings

HIGH

[H-1] Incorrect calculation in `TSwapPool::getInputAmountBasedOnOutput` causes user to pay way more than expected

Description: The formula used to calculate the input token amount for a given output token amount is flawed. The intention is to apply a 0.3% protocol fee (commonly implemented as a 0.997 multiplier on the input), but the calculation mistakenly inverts the fee logic. The formula:

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

multiplies by 10000 and divides by 997, which unintentionally scales the input cost by ~10x. This results in users paying substantially more input tokens than required.

Impact: Users are overcharged when swapping tokens. This not only leads to direct financial loss for users but also undermines trust in the protocol and its token pricing logic. Arbitrageurs may exploit the faulty pricing mechanism, draining liquidity and destabilizing the pool.

Recommended Mitigation:

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-           return ((inputReserves * outputAmount) * 10_000) /
((outputReserves - outputAmount) * 997);
+           return ((inputReserves * outputAmount) * 1_000) /
((outputReserves - outputAmount) * 997);
    }
```

[H-2] No slippage check in `TSwapPool::swapExactOutput` can result loss of funds

Description: The is no parameter like `maxInputToken` which is intended to use to protect from slippage of funds. For example, the use want 1 weth as output token but don't want to spend much input Tokens on it so he should set a limit to the maximum input token he want to spend in order to but a output token but that parameter is missing in this function forcing user to pay the amount which he don't intend to pay.

Impact: The user will have to pay more than his expectations which he don't intend to.

Proof of Concept:

```
function test_swapExactOutput_overpays() public {
    // Deploy mock ERC20 tokens
    MockERC20 tokenA = new MockERC20("TokenA", "TKA");
    MockERC20 tokenB = new MockERC20("TokenB", "TKB");

    // Deploy swap contract
    SwapContract swap = new SwapContract();

    // Provide liquidity: balanced reserves initially
    tokenA.mint(address(swap), 1000 ether);
    tokenB.mint(address(swap), 1 ether); // Drained pool: low output
reserve
```

```
    // Mint user balance
    tokenA.mint(address(this), 1000 ether);
    tokenA.approve(address(swap), type(uint256).max);

    // User wants 1 TokenB — expects to pay small amount of TokenA
    // But due to imbalance, inputAmount will be massive
    uint256 inputUsed = swap.swapExactOutput(
        tokenA,
        tokenB,
        1 ether,    // exact output
        uint64(block.timestamp + 1000)
    );

    emit log_named_uint("Input Paid", inputUsed);

    // Fails expectation: input amount is enormous
    assertGt(inputUsed, 500 ether); // just for illustration
}
```

Recommended Mitigation: Add an erro on in the error section `TSwapPool__InputTooHigh(uint256 actual, uint256 max)`

```

    function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
        uint64 deadline
+       uint256 maxInputToken

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

        //@audit-HIGH there's no check for maximum input tokens
        //IMPACT-HIGH the user may get to pay more than he expected
        //e.x. 1 weth may cost him around 100_000 of pool tokens which he
may not intend to swap
        //LIKELYHOOD-HIGH
        inputAmount = getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

+        if(maxInputToken < inputAmount){
+           Revert TSwapPool__InputTooHigh(inputAmount, maxInputAmount);
+        }

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

[H-3] `TSwapPool::sellPoolTokens` uses a wrong function which causes users to recieve incorrect tokens

Description: The `sellPoolTokens` function is intended to allow users 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 miscalculaes 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 protcol functionality.

Proof of Concept: 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,
minWethToReceive, uint64(block.timestamp));
    }
```

[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 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 follow 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:

```
    function testInvariantBreaks() 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);
        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) - 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++;
-        // Fee-on-transfer
-        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 has a parameter `deadline` and according to the documentation it is used to for the transaction to be completed by. But, it's never used causing to add liquidity even after the dealine has passed.

Impact: Transaction could be sent when market conditions are unfavourable to deposit, even when adding a deadline parameter.

Proof of Concept: The `deadline` parameter is being unused.

Recommended Mitigation:

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

Low

[L-1] The event `TSwapPool::LiquidityAdded` is being emitted incorrectly causing the information causing to misinterpret the data.

Description: The `LiquidityAdded` event is emitted with arguments in the wrong order. It currently emits:

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

However, according to the event definition:

```
event LiquidityAdded(address indexed user, uint256 wethAmount, uint256
poolTokenAmount);
```

Impact: The incorrect ordering causes off-chain systems like subgraphs, analytics dashboards, and dApp UIs to display inaccurate liquidity addition data. Users and developers relying on emitted events for insights or calculations may get misleading information, which could impact decisions and trust.

Recommended Mitigation:

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

[L-2] The `TSwapPool::swapExactInput` returns a constant value misleading the caller

Description: The function `TSwapPool::Input` is intended to return the amount of output tokens but the the `output` is never updated and it returns zero which can mislead the caller.

Impact: The user gets wrong idea of the output tokens that he intends to recieve

Proof of Concept:

Recommended Mitigation:

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )   //@written-info this should be marked as external
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        //@written-low does not return anything
        returns (uint256 output)
```

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

-        uint256 outputAmount = getOutputAmountBasedOnInput(
+        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, output, outputAmount);
    }
```

Informationals

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

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

[I-2] There should be a check for zero address in `PoolFactory` constuctor.

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

        i_wethToken = wethToken;
    }
```

[I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`

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

[I-4]: Define and use `constant` variables instead of using literals

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

▶ 4 Found Instances

- Found in src/TSwapPool.sol Line: 276

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

- Found in src/TSwapPool.sol Line: 295

  ```
              ((outputReserves - outputAmount) * 997);
  ```

- Found in src/TSwapPool.sol Line: 454

  ```
                  1e18,
  ```

- Found in src/TSwapPool.sol Line: 463

  ```
                  1e18,
  ```

[I-5] natspec for the `TSwapPool::swapExactInput` is missing which creates misleading assumptions about the working of function

Dev team must provide suitable description about the utitlity of the function in order to understand the function and test it for any bugs.