Protocol Audit Report

Version 1.0

Prepared by: yappy-yum

July 10, 2025

Contents

Protocol Summary

TSWAP is a 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

yappy-yum 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 yappy-yum 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
Likelihood	High	H	H/M	M
	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

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

```
1ec3c30253423eb4199827f59cf564cc575b46db
```

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

severity	Number of issue found
High	3
Medium	3
Low	2
Informational	4
Gas	1
Total	**13**

Findings

High

[H-1] Erroneous function calls in `TSwapPool::sellPoolTokens`

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(
+           swapExactInput(
                i_poolToken,
+               poolTokenAmount,
                i_wethToken,
-               poolTokenAmount,
+               minWethToReceive,
                uint64(block.timestamp)
            );
    }
```

[H-2] miscalculates fees in `TSwapPool::getInputAmountBasedOnOutput`

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) /
+           ((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.

```
    function test_errorneous_fees_calculation() public {
        // provide starting liquidity
        // -> 1:1 liquidity
        // -> 200 for both weth and token
        vm.startPrank(liquidityProvider);
        console.log("LP total pooltoken balance: ", poolToken.balanceOf(
            liquidityProvider));
        console.log("LP total weth balance: ", weth.balanceOf(
            liquidityProvider));
        console.log("LP total shares before deposits: ", pool.balanceOf(
            liquidityProvider));

        pool.deposit({
            wethToDeposit: 200e18,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: 200e18,
            deadline: uint64(block.timestamp)
        });

        console.log("LP total pooltoken balance before user swap: ",
            poolToken.balanceOf(liquidityProvider));
        console.log("LP total weth balance before user swap: ", weth.
            balanceOf(liquidityProvider));
        console.log("LP total shares after deposits: ", pool.balanceOf(
            liquidityProvider));
        vm.stopPrank();

        // user buy (swap) 1 weth using, using his existing pool token
        // currently, user balance has 11 token before swap
        vm.startPrank(user);
        console.log("user total pooltoken balance before purchase: ",
            poolToken.balanceOf(user));
        console.log("user total weth balance before purchase: ", weth.
            balanceOf(user));

        pool.swapExactOutput({
            inputToken: poolToken,
            outputToken: weth,
            outputAmount: 1 ether,
            deadline: uint64(block.timestamp)
        });

        // Note:
        // initial liquidity was 1:1 ...
        // therefore expecting about ~1 pooltoken be paid
        // however, it spent too much that user balance is not below 1
            ether
        console.log("user total pooltoken balance after purchase: ",
            poolToken.balanceOf(user));
        console.log("user total weth balance after purchase: ", weth.
            balanceOf(user));
        vm.stopPrank();

        // liquidity provider withdraw
        vm.startPrank(liquidityProvider);
        pool.withdraw({
            liquidityTokensToBurn: pool.balanceOf(liquidityProvider),
            minWethToWithdraw: 1,
            minPoolTokensToWithdraw: 1,
            deadline: uint64(block.timestamp)
        });

        // Note:
        // because of these, liquidity provider can rug all funds from the pool ...
        // including those deposited by user.
        console.log("LP total pooltoken balance after withdraw: ", poolToken.balanceOf(liquidityProvider));
        console.log("LP total weth balance after withdraw: ", weth.balanceOf(liquidityProvider));
        console.log("LP total shares after withdraw: ", pool.balanceOf(liquidityProvider));
        vm.stopPrank();
    }
```

below is the console output log to show their balances from the test above:

```
Logs:
  LP total pooltoken balance:  200000000000000000000  (200 ether)
  LP total weth balance:  200000000000000000000       (200 ether)
  LP total shares before deposits:  0                 (0   ether)

  LP total pooltoken balance before user swap:  0         (0   ether)
  LP total weth balance before user swap:  0              (0   ether)
  LP total shares after deposits:  200000000000000000000  (200 ether)

  user total pooltoken balance before purchase:  11000000000000000000 (11 ether)
  user total weth balance before purchase:  0                         (0 ether)

  user total pooltoken balance after purchase:  919507265515138380  (~ 0.92 ether)
  user total weth balance after purchase:  1000000000000000000      (  1 ether)

  LP total pooltoken balance after withdraw:  210080492734484861620 (~ 210 ether)
  LP total weth balance after withdraw:  199000000000000000000      (  199 ether)
  LP total shares after withdraw:  0                                (  0 ether)
```

[H-3] Additional rewards may break invariant

in `TSwapPool::_swap` function, extra tokens (incentive) is given to every 10th caller who does the token swaps. However, this rewards may potentially break the invariant of the constant AMM math

consider removing the additional rewards if its unnecessary

```
-   uint256 private swap_count = 0;
-   uint256 private constant SWAP_COUNT_MAX = 10;

    .
    .
    .

    function _swap(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 outputAmount
    ) private {
        if (
            _isUnknown(inputToken) ||
            _isUnknown(outputToken) ||
            inputToken == outputToken
        ) {
            revert TSwapPool__InvalidToken();
        }

-       swap_count++;
-       if (swap_count >= SWAP_COUNT_MAX) {
-           swap_count = 0;
-           outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
-       }
        emit Swap(
            msg.sender,
            inputToken,
            inputAmount,
            outputToken,
            outputAmount
        );

        inputToken.safeTransferFrom(msg.sender, address(this), inputAmount);
        outputToken.safeTransfer(msg.sender, outputAmount);
    }
```

Medium

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

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.

[M-2] Missing deadline check when adding liquidity

The `deposit` function accepts a `deadline` parameter, which according to documentation is **“he 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 `TSwapPool::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.

```
    error TSwapPool__OutputTooHigh(uint256 actual, uint256 max);

    .
    .
    .
    .

    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 `TSwapPool::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.

```
    function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
    ) private {
        _mint(msg.sender, liquidityTokensToMint);
-       emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
+       emit LiquidityAdded(msg.sender, wethToDeposit, poolTokensToDeposit);

        // Interactions
        i_wethToken.safeTransferFrom(msg.sender, address(this), wethToDeposit);
        i_poolToken.safeTransferFrom(
            msg.sender,
            address(this),
            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] Unused event in `PoolFactory` contract

event `PoolFactory__PoolDoesNotExist` is not used in the `PoolFactory` contract, consider remove it if its indeed unused in this contract.

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

[I-2] missing zero-address checks

while this is not a big deal, an additional safety checks will reduce the chance of going wrong during the deployment.

1. `PoolFactory` constructor:

```
+   error PoolFactory__PoolAlreadyExists(address tokenAddress);

    .
    .
    .

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

2. `TSwapPool::getOutputAmountBasedOnInput`

```
    function getOutputAmountBasedOnInput(
        uint256 inputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(inputAmount)
+       revertIfZero(inputReserves)
        revertIfZero(outputReserves)
        returns (uint256 outputAmount)
    {
```

[I-3] fetch incorrect data in `PoolFactory::createPool`

As the local variable `liquidityTokenSymbol` suggest, it should get the token symbol, rather than token name. Additionally, token name has been fetched for `liquidityTokenName`, `liquidityTokenSymbol` should ideally gets the token symbol instead.

```
    function createPool(address tokenAddress) external returns (address) {
        if (s_pools[tokenAddress] != address(0)) {
            revert PoolFactory__PoolAlreadyExists(tokenAddress);
        }

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

[I-4] Consider External Visibility for Getter

`public` visibility is declared in the `TSwapPool::totalLiquidityTokenSupply`. It is a best practice to use `external` for getter function

```
    /// @notice a more verbose way of getting the total supply of liquidity tokens
-   function totalLiquidityTokenSupply() public view returns (uint256) {
+   function totalLiquidityTokenSupply() external view returns (uint256) {
        return totalSupply();
    }
```

Gas

[G-1] unuse declared local variable in `TSwapPool::deposit`

local variable `poolTokenReserves` is declared, but it is not used within this function. If this variable is intended for comment notes, consider comment this local variable as well for more gas efficient

```
    {
        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));
+           uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
            // Our invariant says weth, poolTokens, and liquidity tokens must always have the same ratio after the
            // initial deposit
            // poolTokens / constant(k) = weth
            // weth / constant(k) = liquidityTokens
            // aka...
            // weth / poolTokens = constant(k)
            // To make sure this holds, we can make sure the new balance will match the old balance
            // (wethReserves + wethToDeposit) / (poolTokenReserves + poolTokensToDeposit) = constant(k)
            // (wethReserves + wethToDeposit) / (poolTokenReserves + poolTokensToDeposit) =
            // (wethReserves / poolTokenReserves)
            //
            // So we can do some elementary math now to figure out poolTokensToDeposit...
            // (wethReserves + wethToDeposit) = (poolTokenReserves + poolTokensToDeposit) * (wethReserves / poolTokenReserves)
            // wethReserves + wethToDeposit  = poolTokenReserves * (wethReserves / poolTokenReserves) + poolTokensToDeposit * (wethReserves / poolTokenReserves)
            // wethReserves + wethToDeposit = wethReserves + poolTokensToDeposit * (wethReserves / poolTokenReserves)
            // wethToDeposit / (wethReserves / poolTokenReserves) = poolTokensToDeposit
            // (wethToDeposit * poolTokenReserves) / wethReserves = poolTokensToDeposit
            uint256 poolTokensToDeposit = getPoolTokensToDepositBasedOnWeth(
                wethToDeposit
            );
            if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
                revert TSwapPool__MaxPoolTokenDepositTooHigh(
                    maximumPoolTokensToDeposit,
                    poolTokensToDeposit
                );
            }
```