TSwap Protocol Audit Report

Version 1.0

Norbert Orgován

January 29, 2024

Protocol Audit Report

Norbert Orgován

January 29, 2024

Prepared by: Orgovan & Churros

Lead Auditors: - Norbert Orgovan

Table of Contents

Protocol Summary

The TSwap Protocol implements a decentralized exchange. The codebase is a modified version of the Uniswap v1 codebase. T-Swap is known as an Automated Market Maker (AMM) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset.

Disclaimer

The Orgovan & Churros 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
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

Scope

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda
- In Scope:

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

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

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

We had 1 expert auditor assigned to this audit who spent xxx hours to thouroughly review the PuppyRaffle codebase. Using both manual review and a number of tools (e.g. Slyther, Aderyn), a number of issues have been found, as detailed below.

Severity	Number of issues found
High	5
Medium	2
Low	2
Informational	11
Gas	1
Total	21

Issues found

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to overtax users with a 90.3% fee

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens the users should deposit given the amount 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, resulting in a 90.3% fee.

Impact: Protocol takes more fees than expected by the users.

Proof of Concept: Consider the following scenario:

1. The user calls `TSwapPool::swapExactOutput` to buy a predefined amount of output tokens in exchange for an undefined amount of input tokens.
2. `TSwapPool::swapExactOutput` calls the `getInputAmountBasedOnOutput` function that is supposed to calculate the amount of input tokens required to result in the predefined amount of output tokens. However, the fee calculation in this function in incorrect.
3. The user gets overtaxed with a 90.4% fee.

Proof of Code:

Code

```
1
2  function test_overTaxingUsersInSwapExactOutput() public {
3          // providing liquidity to the pool
4          uint256 initialLiquidity = 100e18;
5
6          vm.startPrank(liquidityProvider);
7          weth.approve(address(pool), initialLiquidity);
8          poolToken.approve(address(pool), initialLiquidity);
9
10         pool.deposit({
11             wethToDeposit: initialLiquidity,
12             minimumLiquidityTokensToMint: 0,
13             maximumPoolTokensToDeposit: initialLiquidity,
14             deadline: uint64(block.timestamp)
15         });
16         vm.stopPrank();
17
```

```
18          // @audit this function actually returns an incorrect value
19          // uint256 priceOfOneWeth = pool.getPriceOfOneWethInPoolTokens
                ();
20
21          address someUser = makeAddr("someUser");
22          uint256 userInitialPoolTokenBalance = 11e18;
23          poolToken.mint(someUser, 11e18); // now the user has 11 pool
                tokens
24
25          // user intends to buy 1 weth with pool tokens
26          vm.startPrank(someUser);
27          poolToken.approve(address(pool), type(uint256).max);
28          pool.swapExactOutput(poolToken, weth, 1e18, uint64(block.
                timestamp));
29          vm.stopPrank();
30
31          uint256 userEndingPoolTokenBalance = poolToken.balanceOf(
                someUser);
32
33          console.log("Initial user balance: ",
                userInitialPoolTokenBalance);
34          // console.log("Price of 1 weth: ", priceOfOneWeth);
35          console.log("Ending user balance: ", userEndingPoolTokenBalance
                );
36
37          // Initial liquidity was 1:1, so user should have paid ~1 pool
                token
38          // However, it spent much more than that. The user started with
                 11 tokens, and now only has less than 1.
39          assert(userEndingPoolTokenBalance < 1 ether);
40      }
```

Recommended Mitigation:

```
1       function getInputAmountBasedOnOutput(
2           uint256 outputAmount,
3           uint256 inputReserves,
4           uint256 outputReserves
5       )
6           public
7           pure
8           revertIfZero(outputAmount)
9           revertIfZero(outputReserves)
10          returns (uint256 inputAmount)
11      {
12  -           return ((inputReserves * outputAmount) * 10_000) / ((
        outputReserves - outputAmount) * 997);
13  +           return ((inputReserves * outputAmount) * 1_000) / ((
        outputReserves - outputAmount) * 997);
14
15      }
```

[H-2] Lack of slippage protection in `TswapPool::swapExactOutput` casues users to potentially receive way fewer tokens

Description: The funxtion `swapExactOutput` does not include any kind of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput`, where the function specifies the `minOutputAmount`. Similarly, `swapExactOutput` should specify a `maxInputAmount`.

Impact: If the market conditions change before the transaction process, the user could get a much worse swap then expected.

Proof of Concept:

1. The price of WETH is 1_000 USDC.
2. User calls `swapExactOutput`, looking for 1 WETH with the following parameters:
    - inputToken: USDC
    - outputToken: WETH
    - outputAmount: 1
    - deadline: whatever
3. The function does not allow a `maxInputAmount`.
4. As the transaction is pending in the mempool, the market changes, and the price movement is huge: 1 WETH now costs 10_000 USDC, 10x more than the user expected!
5. The transaction completes, but the user got charged 10_000 USDC for 1 WETH.

Recommended Mitigation: Include a `maxInputAmount` input parameter in the function declaration, so that the user could specify the maximum amount of tokens he would like to spend and, hence, could predict their spending when using this function of the protocol.

```
    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);
```

```
18
19 +        if(inputAmount > maxInputAmount){
20 +            revert();
21 +        }
22
23          _swap(inputToken, inputAmount, outputToken, outputAmount);
24      }
```

[H-3] `TSwapPool::sellPoolTokens` mistakenly calls the incorrect swap function, causing users to receive the incorrect amount of tokens

Description: The `sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. In the `poolTokenAmount` parameter, users indicate how many pool token they intend to sell. However, the function mistakenly calls `swapExactOutput` instead of `swapExactInput` to perform the swap, and therein assignes the value of `poolTokenAmount` to function input argument `outputAmount`, effectively mixing up the input and output tokens / amounts.

Impact: Users will swap the incorrects amount of tokens, which severely discrupts the functionality of the protocol.

Proof of Concept: Consider the following scenario:

1. A user has 100 pool tokens, and wants to sell 5 by calling the `sellPoolTokens` function.
2. Instead of the `swapExactInput` function, `sellPoolTokens` calls `swapExactOutput`.
3. In `swapExactOutput`, `poolTokenAmount` is used as `outputAmount` while it is really the input amount.
4. As a result, user will swap more output tokens than originally intended.

Apart from this, the user will be overtaxed due to a bug in `getInputAmountBasedOnOutput()` called by `swapExactOutput`.

Proof of Code: Add this piece of code `TSwapPool.t.sol`:

Code

```
1 function test_sellPoolTokensCallsTheIncorrectSwapFunction() public {
2         // setting up the pool by providing liquidity
3         uint256 initialLiquidity = 100e18;
4
5         vm.startPrank(liquidityProvider);
6         weth.approve(address(pool), initialLiquidity);
7         poolToken.approve(address(pool), initialLiquidity);
8
9         pool.deposit({
```

```
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // setting up the user
        address someUser = makeAddr("someUser");
        uint256 userStartingPoolTokenBalance = 100 ether;
        poolToken.mint(someUser, userStartingPoolTokenBalance);
        vm.prank(someUser);
        poolToken.approve(address(pool), type(uint256).max);

        // user intends to sell 5 pool tokens
        vm.prank(someUser);
        uint256 poolTokensToSell = 5e18;
        // @note that sellPoolTokens() uses swapExactOutput() to perform the swap,
        // which in turn calls getInputAmountBasedOnOutput() to calculate the amount of input tokens to be
        // deducted from the user, and this function miscalculates the fee, so to make things worse,
        // the user becomes subject of overtaxing too
        pool.sellPoolTokens(poolTokensToSell);

        uint256 expectedEndingUserPoolTokenBalance = userStartingPoolTokenBalance - poolTokensToSell;
        uint256 realEndingUserPoolTokenBalance = poolToken.balanceOf(someUser);

        console.log("Expected pool token balance of the user: ", expectedEndingUserPoolTokenBalance);
        console.log("Real pool token balance of the user: ", realEndingUserPoolTokenBalance);

        assert(expectedEndingUserPoolTokenBalance > realEndingUserPoolTokenBalance);
    }
```

Recommended Mitigation: Change the implementation to use `swapExactInput` instead of the `swapExactOutput` function. Note that this would require the `sellPoolTokens` function to accept an additional parameter (i.e. `minOutputAmount` to be passed to `swapExactInput`).

```
-    function sellPoolTokens(uint256 poolTokenAmount) external returns (uint256 wethAmount) {
+    function sellPoolTokens(uint256 poolTokenAmount, uint256 minWethToReceive) external returns (uint256 wethAmount) {

```

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

     }
```

Additionally, it might be wise to add a deadline to the function, as currently there is no deadline. MEV later.

[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 in the pool - `y`: The balance of WETH in the pool - `k`: The constant product of the 2 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 (a full extra token after every 10 swaps) in the `_swap` function, meaning that over time the protocol funds would 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 (a full extra token after every 10 swaps) given out by the protocol.

More simply put, the core invariant of the protocol is broken!

Proof of Concept: 1. A user swaps 10 times and collects the extra incentive of 1 token (1 `_000_000_000_000_000_000`) 2. The usercontinues to swap until all the protocol funds are drained.

Proof of Code:

Code

```
function test_InvariantBreaks() public {
        // providing liquidity
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
```

```
5           poolToken.approve(address(pool), 100e18);
6           pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
7           vm.stopPrank();
8
9           uint256 outputWeth = 1e17;
10
11          // set up user, than perform 9 swaps
12          vm.startPrank(user);
13          poolToken.mint(user, 100e18);
14          poolToken.approve(address(pool), type(uint256).max);
15          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
16          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
17          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          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
24
25          // from the invariant test (the handler).
26          // We use these here because the interesting thing happens at
                the 10th swap
27          int256 startingY = int256(weth.balanceOf(address(pool)));
28          int256 expectedDeltaY = int256(-1) * int256(outputWeth); //
                this is deltaY
29
30          // and then do a swap for the 10th time
31          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
32          vm.stopPrank();
33
34          // from the invariant test (the handler)
35          uint256 endingY = weth.balanceOf(address(pool));
36          int256 actualDeltaY = int256(endingY) - int256(startingY); //
                this could be negative
37
38          assertEq(expectedDeltaY, actualDeltaY);
39      }
```

Recommended Mitigation: Remove the extra incentive mechanism. If you want to keep this nonetheless, you should account for the change in the `x * y = k` invariant. Alternatively, you could set aside

tokens the same way you did with fees.

```
-         swap_count++;
-         if (swap_count >= SWAP_COUNT_MAX) {
-             swap_count = 0;
-             outputToken.safeTransfer(msg.sender, 1
    _000_000_000_000_000_000);
-         }
```

[H-5] Rebase, fee-on-transfer, and ERC777 tokens break protocol invariant

Description: Weird ERC20 tokens with uncommon / malicious implementations can endanger the whole protocol. Examples include rebase, fee-on-transfer, and ERC777 tokens.

Impact:

Proof of Concept:

Recommended Mitigation:

Medium

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

Description: The `deposit` function accepts a `deadline` as an input the parameter which, according to the documentation, "the deadline for the transaction to be completed by". However, this parameter is never actually used. As a consequence, operations that add liquidty to the pool might be executed at unexpected times, in market conditions when the deposit rate is unfavorable.

This also makes this part susceptible to MEV attacks.

Impact: Transactions can be sent when market conditions are unfavorable, even when the deadline is set.

Proof of Concept: The `deadline` parameter is unused (this is highlighted by the compiler too).

Recommended Mitigation: Make the following change to the function:

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

```
8 +         revertIfDeadlinePassed(deadline)
9           revertIfZero(wethToDeposit)
10          returns (uint256 liquidityTokensToMint)
11      {...}
```

[M-2] `TSwapPool::getPriceOfOneWethInPoolTokens()` and `TSwapPool::getPriceOfOnePoolTokenInWeth()` return incorrect price values

Description: `getPriceOfOneWethInPoolTokens` is supposed to return the price of 1 WETH in terms of pool tokens, and `TSwapPool::getPriceOfOnePoolTokenInWeth` is supposed to return the price of 1 pool token in terms of WETH. However, the return values are incorrect. Both functions return the amount of output tokens after fees, which is not the same as the price of 1 output token in input tokens. (Consider this: as compared to a fee-less protocol, if there are fees, the amount of output tokens should be lower, while the price should be not lower but higher.)

Impact: User will think that the WETH / pool token is cheaper that it actually is, and they might make their trading decisions based on this incorrect price information. E.g. they might think the price of their token is falling, might panic and sell their tokens to avoid further losses by calling `sellPoolTokens()`.

Proof of Concept: Consider the following scenario:

1. A user has 1 WETH, and wants to swap it for pool tokens.
2. The user calls `getPriceOfOneWethInPoolTokens` and sees an incorrect price that is the inverse of the actual price.
3. User finds the price appealing and swaps his WETH.
4. User ends up with a lot less pool tokens than he expected.

Proof of Code: Insert this piece of code to `TSwapPool.t.sol` (note the it demonstrates a different scenario than the one written under "Proof of Concept"):

Code

```
1  /**
2       * @notice In scenarios where inputAmount is close to the
            minOutputAmount, even a small loss of precision can lead
3       * to the outputAmount falling below minOutputAmount, triggering
            the TSwapPool__OutputTooLow error.
4       */
5      function
           test_incorrectPriceValueReturnedByGetPriceOfOneWethInPoolTokens
           () public {
6          uint256 precision = 1 ether;
7
```

```
8           // we need more liquidity in the pool, so granting additional
               money for the provider
9           weth.mint(liquidityProvider, 800e18);
10          poolToken.mint(liquidityProvider, 800e18);
11
12          // providing liquidity to the pool
13          uint256 initialLiquidity = 1000e18;
14
15          vm.startPrank(liquidityProvider);
16          weth.approve(address(pool), initialLiquidity);
17          poolToken.approve(address(pool), initialLiquidity);
18
19          pool.deposit({
20              wethToDeposit: initialLiquidity,
21              minimumLiquidityTokensToMint: 0,
22              maximumPoolTokensToDeposit: initialLiquidity,
23              deadline: uint64(block.timestamp)
24          });
25          vm.stopPrank();
26
27          uint256 incorrectPriceOfOneWeth = pool.
               getPriceOfOneWethInPoolTokens();
28          uint256 correctPriceOfOneWeth = (1e18 * precision) /
               incorrectPriceOfOneWeth;
29
30          console.log("Incorrect price: ", incorrectPriceOfOneWeth); //
               987_158_034_397_061_298 = 9.87*10**17
31          console.log("Correct price: ", correctPriceOfOneWeth);
32
33          address userSuccess = makeAddr("userSuccess");
34          address userFail = makeAddr("userFail");
35
36          // userFail attempts to buy 1 weth with a balance of pool
               tokens that equals the incorrect price of 1 weth
37          poolToken.mint(userFail, incorrectPriceOfOneWeth);
38          vm.startPrank(userFail);
39          poolToken.approve(address(pool), type(uint256).max);
40          // expect a revert (specifically, TSwapPool__OutputTooLow)
41          vm.expectRevert();
42          // using swapExactOutput() would be more appropriate here, but
               that one has a huge bug
43          pool.swapExactInput({
44              inputToken: poolToken,
45              inputAmount: incorrectPriceOfOneWeth,
46              outputToken: weth,
47              minOutputAmount: 99999e13, // due to precision loss, we
                   cant really expect to get 1 full weth (1000e15), we
48                  // can only approximate
49              deadline: uint64(block.timestamp)
50          });
51          vm.stopPrank();
```

```
52
53          // userSuccess attempts to buy 1 weth with a balance of pool
                tokens that equals the correct price of 1 weth
54          poolToken.mint(userSuccess, correctPriceOfOneWeth);
55          vm.startPrank(userSuccess);
56          poolToken.approve(address(pool), type(uint256).max);
57          // using swapExactOutput() would be more appropriate here, but
                that one has a huge bug
58          pool.swapExactInput({
59              inputToken: poolToken,
60              inputAmount: correctPriceOfOneWeth,
61              outputToken: weth,
62              minOutputAmount: 99999e13, // due to precision loss, we
                    cant really expect to get 1 full weth (1000e15), we
63                  // can only approximate
64              deadline: uint64(block.timestamp)
65          });
66          vm.stopPrank();
67
68          assert(weth.balanceOf(userSuccess) > 999e15); // has nearly 1
                full weth
69          assertEq(poolToken.balanceOf(userSuccess), 0); // spent all his
                poolToken
70      }
```

Recommended Mitigation:

```
1       function getPriceOfOneWethInPoolTokens() external view returns (
            uint256) {
2   -         return getOutputAmountBasedOnInput(
3   -             1e18, i_wethToken.balanceOf(address(this)), i_poolToken.
        balanceOf(address(this))
4   -         );
5   +         uint256 precision = 1e18;
6   +         uint256 amountOfPoolTokensReceivedForOneWeth =
        getOutputAmountBasedOnInput(
7   +             1e18, i_wethToken.balanceOf(address(this)), i_poolToken.
        balanceOf(address(this)));
8   +
9   +         uint256 priceOfOneWethInPoolTokens = (1e18 * precision) /
        amountOfPoolTokensReceivedForOneWeth;
10  +
11  +       return priceOfOneWethInPoolTokens;
12      }
13
14      function getPriceOfOnePoolTokenInWeth() external view returns (
            uint256) {
15  -         return getOutputAmountBasedOnInput(
16  -             1e18, i_poolToken.balanceOf(address(this)), i_wethToken.
        balanceOf(address(this))
17  -         );
```

```
18 +       uint256 precision = 1e18;
19 +       uint256 amountOfWethReceivedForOnePoolToken =
   getOutputAmountBasedOnInput(
20 +           1e18, i_poolToken.balanceOf(address(this)), i_wethToken.
   balanceOf(address(this)));
21 +
22 +        uint256 priceOfOnePoolTokenInWeth = (1e18 * precision) /
   amountOfWethReceivedForOnePoolToke;
23 +
24 +     return priceOfOnePoolTokenInWeth;
25      }
```

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order, causing event to emit in'correct information

Description: When the `LiquidtyAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTrans` , it logs values in the incorrect order. The `poolTokensToDeposit` value should go to the 3rd parameter position, whereas the `wethToDeposit` should go to 2nd.

Impact: The emit emission is incorrect, causing off-chain functions potentially malfunctioning.

Recommended Mitigation:

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

[L-2] In the function declaration of `TSwapPool::swapExactInput`, a return value is defined but never assigned a value, resulting in a default but incorrect return value

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the user. However, while it declares the named return value `output`, it never assigns a value to it, nor uses an explicit return statement.

Impact: The return value will always be 0, giving an incorrect information to the user.

Proof of Concept:

Recommended Mitigation:

```
1     function swapExactInput(
2         IERC20 inputToken, // e input token to swap, e.g. sell DAI
```

```
3           uint256 inputAmount, // e amount of DAI to sell
4           IERC20 outputToken, // e token to buy, e.g. weth
5           uint256 minOutputAmount, // mint weth to get
6           uint64 deadline // deadline for the swap execution
7       )
8           public
9           revertIfZero(inputAmount)
10          revertIfDeadlinePassed(deadline)
11          returns (
12  -           uint256 output
13  +           uint256 outputAmount
14
15          )
16      {
17          uint256 inputReserves = inputToken.balanceOf(address(this));
18          uint256 outputReserves = outputToken.balanceOf(address(this));
19
20  -        uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount
        , inputReserves, outputReserves);
21  +        outputAmount = getOutputAmountBasedOnInput(inputAmount,
        inputReserves, outputReserves);
22
23
24
25          if (outputAmount < minOutputAmount) {
26              revert TSwapPool__OutputTooLow(outputAmount,
                    minOutputAmount);
27
28
29          }
30          _swap(inputToken, inputAmount, outputToken, outputAmount);
31
32  +       return outputAmount;
33
34      }
```

Informatinals

[I-1] Custom error `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` is not used

Description: Custom error `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` is not used anywhere in the code, and should be removed.

Recommended Mitigation:

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

[I-2] Missing zero address checks in both the `PoolFactory.sol` and `TSwapPool.sol` contracts

Description: Both `PoolFactory.sol` and `TSwapPool.sol` contain functions which do not check if the address provided as an input argument is the zero address (address(0)). According to the best practice, zero address checks should be performed. Instantes:

- `contructor(address wethToken)` in `PoolFactory.sol`
- `constructor(address poolToken, address wethToken, string memory liquidityTokenName, string memory liquidityTokenSymbol)` in `TSwapPool.sol`

Recommended Mitigation: Add checks to these functions, preferably with custom error. Example for the first instance

```
+       error PoolFactory__IsZeroAddress();

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

[I-3] Use `.symbol` instead of `.name` when assigning a value to `liquidityTokenSymbol` In `PoolFactory::createPool()`,

Description: To be more in line with the name of the variable (and intended use thereof), use `.symbol` instead of `.name` when assigning a value to `liquidityTokenSymbol` In `PoolFactory::createPool()`

Recommended Mitigation:

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

[I-4] Consider indexing up to 3 input parameters in events for facilitating better searching and filtering

Description: Indexing is a feature used with events. When you declare an event, you can mark up to three parameters as indexed. This means these parameters are treated in a special way by the

Ethereum Virtual Machine (EVM). Indexing facilitates searching and filtering: when parameters in an event are indexed, they are stored in a way that allows you to search and filter for these events using these parameters. Indexing is commonly used in scenarios like tracking transfers of tokens, logging changes in ownership, or recording key actions taken in a contract. At the same time, however, keep in mind that you can index only up to 3 parameters, and that indexing increases the gas cost of event emission.

Found in:

- src/PoolFactory.sol Line: 35
- src/TSwapPool.sol Line: 43
- src/TSwapPool.sol Line: 44
- src/TSwapPool.sol Line: 45

Recommended Mitigation:

```
1 -     event Swap(address indexed swapper, IERC20 tokenIn, uint256
    amountTokenIn, IERC20 tokenOut, uint256 amountTokenOut);
2 +     event Swap(address indexed swapper, indexed IERC20 tokenIn,
    indexed uint256 amountTokenIn, IERC20 tokenOut, uint256
    amountTokenOut);
```

[I-5] No need to emit public constant `MINIMUM_WETH_LIQUIDITY` in custom error `TSwapPool::TSwapPool__WethDepositAmountTooLow`

Description: The value of public constant `MINIMUM_WETH_LIQUIDITY` can be queried by anybody any time on the blockchain and, hence, it is unneccesary to emit it in custom error `TSwapPool::TSwapPool__WethDepositAmountTooLow`.

Recommended Mitigation:

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

[I-6] Explanatory documentation in `TSwapPool::deposit` is partially incorrect

Description: The following part of the `TSwapPool:deposit` documentation is incorrect:

” // So we can do some elementary math now to figure out poolTokensToDeposit… // (wethReserves + wethToDeposit) / poolTokensToDeposit = wethReserves // (wethReserves + wethToDeposit) = wethReserves * poolTokensToDeposit // (wethReserves + wethToDeposit) / wethReserves = poolTokensToDeposit ”

Note that the parts above these are correct but not this part. The transition to (wethReserves+wethToDeposit)/poolTokensToDeposit=wethReserves is where the confusio arises. This equation does not maintain the constant product formula. It instead suggests that the ratio of the total WETH (after deposit) to the pool tokens to deposit equals the original WETH reserves, which is not consistent with maintaining the constant product.

[I-7] CEI is not followed in `TSwapPool::deposit`

Description: CEI (Checks-Effects-Interaction), the best-practice design pattern used for avoding reentrency attacks, is not followed in `TSwapPool::deposit`.

Code

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
            // @audit-info no need to emit MINIMUM_WETH_LIQUIDITY, as it is a public constant, can be checked any time
            revert TSwapPool__WethDepositAmountTooLow(MINIMUM_WETH_LIQUIDITY, wethToDeposit);
        }
        if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this));
            // @audit gas. This is not used, dont need this line. (Probably you first wanted a manual calculation, but
            // ...ended up using getPoolTokensToDepositBasedOnWeth())
            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)
```

```
28              // (wethReserves + wethToDeposit) / (poolTokenReserves +
                    poolTokensToDeposit) =
29              // (wethReserves / poolTokenReserves)
30
31              // So we can do some elementary math now to figure out
                    poolTokensToDeposit...
32              // (wethReserves + wethToDeposit) / poolTokensToDeposit =
                    wethReserves
33              // (wethReserves + wethToDeposit)  = wethReserves *
                    poolTokensToDeposit
34              // (wethReserves + wethToDeposit) / wethReserves  =
                    poolTokensToDeposit
35              uint256 poolTokensToDeposit =
                    getPoolTokensToDepositBasedOnWeth(wethToDeposit);
36              // e if we calculate too many poolTokens to deposit, revert
                    , good
37              if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
38                  revert TSwapPool__MaxPoolTokenDepositTooHigh(
                        maximumPoolTokensToDeposit, poolTokensToDeposit);
39              }
40
41              // We do the same thing for liquidity tokens. Similar math.
42              liquidityTokensToMint = (wethToDeposit *
                    totalLiquidityTokenSupply()) / wethReserves;
43              if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
44                  revert TSwapPool__MinLiquidityTokensToMintTooLow(
                        minimumLiquidityTokensToMint, liquidityTokensToMint)
                        ;
45              }
46              _addLiquidityMintAndTransfer(wethToDeposit,
                    poolTokensToDeposit, liquidityTokensToMint);
47          } else {
48              // This will be the "initial" funding of the protocol. We
                    are starting from blank here!
49              // We just have them send the tokens in, and we mint
                    liquidity tokens based on the weth
50              _addLiquidityMintAndTransfer(wethToDeposit,
                    maximumPoolTokensToDeposit, wethToDeposit);
51              // e not a state var, but nonetheless
52              // @audit-info move this line before the external calls to
                    follow CEI
53      @>          liquidityTokensToMint = wethToDeposit;
54          }
55      }
```

Impact: Since `liquidityTokensToMint` is not a state variable, it is not that much of an issue that it is being updated after the interactions (making calls to external functions). Still, as a best practice, it is better to you CEI here as well.

Recommended Mitigation:

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

[I-8] Natspec in `TSwapPool::_addLiquidityMintAndTransfer` is incorrect, refers to non-existent function

Description: The natspec in `TSwapPool::_addLiquidityMintAndTransfer` incorrectly refers to a non-existent function `addLiquidity`. There is no such function - instead, the function used to add liquidity to a pool is `deposit`.

[I-9] Use of "magic" numbers (numbers without descriptors) is discouraged

It can be confusing to see number literals in a codebase and it is much more readable if the numbers are given a name. Moreover, using number literals can easily lead to errors, see one of the high vulnerabilities in the report.

Examples:

In `TSwapPool::getAmountBasedOnInput`:

```
        uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * 1000) +
           inputAmountMinusFee;
```

and in `TSwapPool::getAmountBasedOnOutput`:

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

Instead, you could use:

```
uint256 public constant AMOUNT_PRECISION = 1000;
uint256 public constant RETURNED_AMOUNT = 997;
```

[I-10] `TSwapPool:swapExactInput` lacks natspec

Description: No documentation, explanation is provided for key function `swapExactInput`.

[I-11] `TSwapPool:swapExactInput` can be marked as an external function

Description: `swapExactInput` is declared as a public function. However, it is not used internally and, hence, can be declared as external instead.

Gas

[G-1] The `wethReserves` local variable is not used in `TSwapPool::deposit` and, as such, should be removed to save gas.

Description:

Recommended Mitigation:

```
1 -             uint256 wethReserves = i_wethToken.balanceOf(address(this
      ));
```