TSwap Protocol Audit Report

Version 1.0

TrustWarden

October 19, 2024

T-Swap Protocol Audit Report

TrustWarden

October 19, 2024

Prepared by: TrustWarden Lead Auditors:

- TrustWarden

Table of Contents

    * [H-4] In `TSwapPool::_swap` the extra tokens given to users after `swapCount` reach 10, it breaks the protocol invariant of `x * y = k`

- Medium
    * [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline
    * [M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol's invariant
- Low
    * [L-1] `TSwapPool::LiquidityAdded` event has parameters out of order causing event to emit incorrect information
    * [L-2] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given
- Information
    * [I-1] Import `IERC20` interface from forge-std in `PoolFactory` may have some issues
    * [I-2]: Event is missing `indexed` fields
    * [I-3] Lack of zero check on `wethToken` address parameter in `PoolFactory` constructor cause end up to set zero address
    * [I-4] Mistakenly fetching the name of the pool token instead of its symbol
    * [I-5] `PoolFactory__PoolDoesNotExist` error never use in the contract
    * [I-6] `MINUMUM_WETH_LIQUIDITY` is a constant variable and does not need to be emitted from `TSwapPool__WethDepositAmountTooLow` event
    * [I-7] It is better to set `liquidityTokensToMint` before external calls in `TSwapPool::deposit` function
    * [I-8] Lack of zero check in `TSwapPool` constructor, cause to set zero address on WETH and pool token, would break contract
    * [I-9] Missing `revertIfZero` modifier check on `inputReserves` in `TSwapPool::getOutputAmountBasedOnInput`
    * [I-10] Magic numbers in `TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool::getInputAmountBasedOnOutput` functions, might boil down to set wrong numbers in the code
    * [I-11] Lack of inline documentation (NatSpec) in important functions
    * [I-12] Missing `revertIfZero` modifier check on `inputReserves` in `TSwapPool::getInputAmountBasedOnOutput`
    * [I-13] Using literal numbers in `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` functions, magic numbers could be lead to make a mistake
- Gas
    * [G-1] `poolTokenReserves` never used in the `TSwapPool::deposit` function

and will waste gas

* [G-2] The `TSwapPool::swapExactInput` should declare as external it would use more gas

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) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset. It is similar to Uniswap. To understand Uniswap, please watch this video: Uniswap Explained

Disclaimer

The TrustWarden 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

It was quite rough to get understand of protocol's invariant at first and of course to get know well about its equations to maintain the contract functional, and not lose users and also protocol funds. In order to reach this milestone, I spent many hours to understand the meaning behind invariant. It was a great experience and a different approach to find problems in codebase than manual review.

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

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2

Severity	Number of issues found
Info	13
Gas	2
Total	23

Findings

High

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

Description: 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 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Proof of Concept:

1. 100 pool tokens and 50 WETH funded in the pool
2. User wants to swap 10 pool tokens, so he should get 5 WETH
3. The protocol must take %0.3 fees on each swaps

Add these lines of code into `TSwapPool.t.sol`

PoC

```
    function testBreakGetInputAmountBasedOnOutput() public {
        uint256 wethLiquidity = 50e18;
        uint256 poolTokenLiquidity = 100e18;

        vm.startPrank(liquidityProvider);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.deposit(
            wethLiquidity,
            wethLiquidity,
            poolTokenLiquidity,
            uint64(block.timestamp)
```

```
        );
        vm.stopPrank();

        uint256 poolTokenReserve = poolToken.balanceOf(address(pool));
        uint256 wethReserve = weth.balanceOf(address(pool));

        uint256 resultOfGetInput = pool.getInputAmountBasedOnOutput(
            5e18,
            poolTokenReserve,
            wethReserve
        );

        uint256 expectedInputAmount = ((poolTokenReserve * 5e18) * 1
           _000) / ((wethReserve - 5e18) * 997);

        assertEq(resultOfGetInput, expectedInputAmount);
    }
```

Recommended Mitigation: Make change to the function as follows.

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

Description: The `swapExactOutput` function 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 transaction 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 = whatever
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 compeletes, but the user sent the protocol 10,000 USDC instead of the expected 1,000 USDC

- Put this block of code in `TSwapPool.t.sol`
  PoC

```
    function testHugeChangesToSwapRatio() public {
        uint256 wethLiquidity = 50e18;
        uint256 poolTokenLiquidity = 100e18;

        vm.startPrank(liquidityProvider);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.deposit(
            wethLiquidity,
            wethLiquidity,
            poolTokenLiquidity,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        address swapper = makeAddr("swapper");
        poolToken.mint(user, 2000e18);
        weth.mint(user, 2000e18);
        poolToken.mint(swapper, 2000e18);
        weth.mint(swapper, 2000e18);

        uint256 oneWethPriceBeforeHugeSwap = pool
            .getPriceOfOneWethInPoolTokens();

        vm.startPrank(user);
        weth.approve(address(pool), type(uint256).max);
        poolToken.approve(address(pool), type(uint256).max);
        vm.stopPrank();

        vm.startPrank(swapper);
        weth.approve(address(pool), type(uint256).max);
```

```
32          poolToken.approve(address(pool), type(uint256).max);
33          pool.swapExactOutput(poolToken, weth, 10e18, uint64(block.
                timestamp));
34          vm.stopPrank();
35
36          uint256 oneWethPriceAfterHugeSwap = pool
37              .getPriceOfOneWethInPoolTokens();
38
39          uint256 amountWethUserAsk = 1e18;
40          uint256 startUserPLBalance = poolToken.balanceOf(user);
41
42          vm.startPrank(user);
43          pool.swapExactOutput(
44              poolToken,
45              weth,
46              amountWethUserAsk,
47              uint64(block.timestamp)
48          );
49          vm.stopPrank();
50
51          uint256 endUserPLBalance = poolToken.balanceOf(user);
52
53          console.log("In first user swap weth is worth 2 pool tokens");
54          console.log(oneWethPriceBeforeHugeSwap);
55          console.log("Price after first swap happened");
56          console.log(oneWethPriceAfterHugeSwap);
57
58          assertEq(
59              endUserPLBalance,
60              startUserPLBalance + oneWethPriceBeforeHugeSwap
61          );
62      }
```

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 swap.

Add this error in the contract definition

```
1 +     error TSwapPool__InputTooHigh(uint256 actual, uint256 max);
```

And make the following changes in `swapExactOutput`

Code

```
1       function swapExactOutput(
2           IERC20 inputToken,
3 +         uint256 maxInputAmount
4           .
5           .
6           .
7           inputAmount = getInputAmountBasedOnOutput(
```

```
            outputAmount,
            inputReserves,
            outputReserves
        );
+       if (inputAmount > maxInputAmount) {
+           revert(TSwapPool__InputTooHigh(inputAmount, maxInputAmount)
    );
+       }

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

[H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens

Description: The `sellPoolTokens` function 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 miscalculates 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, not output.

Impact: Users will swap the wrong amount of tokens, which is a severe disruption of protocol functionality.

Proof of Concept:

1. A user wants to sell their 1 pool Tokens
2. User Expects to get 0.5 WETH in exchange
3. Since the input parameters order are wrong, causing miscalculation in the amount that needed to exchange perform
4. It'll take +20 pool tokens from user instead of taking 1 pool token
5. The user only get 1 WETH in exchange

Include following block of code in `TSwapPool.t.sol`

Proof of Code

```
    function testBreakSellPoolTokens() public {
        uint256 wethLiquidity = 50e18;
        uint256 poolTokenLiquidity = 100e18;

        vm.startPrank(liquidityProvider);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
```

```
        pool.deposit(
            wethLiquidity,
            wethLiquidity,
            poolTokenLiquidity,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        weth.mint(user, 100e18);
        poolToken.mint(user, 100e18);

        uint256 userStartingWeth = weth.balanceOf(user);
        uint256 wethReserve = weth.balanceOf(address(pool));
        uint256 poolTokenReserve = poolToken.balanceOf(address(pool));

        uint256 userPoolTokenSwapAmount = 1e18;
        uint256 expectedChangeToWeth = pool.getInputAmountBasedOnOutput
            (
            userPoolTokenSwapAmount,
            poolTokenReserve,
            wethReserve
        );

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.sellPoolTokens(userPoolTokenSwapAmount);
        vm.stopPrank();

        uint256 endingPoolTokenReserve = poolToken.balanceOf(address(
            pool));
        uint256 userEndingWeth = weth.balanceOf(user);

        assertEq(endingPoolTokenReserve, poolTokenReserve -
            userPoolTokenSwapAmount);
        assertEq(expectedChangeToWeth, userEndingWeth -
            userStartingWeth);
    }
```

Recommended Mitigation:

Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept 2 new parameters (i.e. `minWethToReveive` and `deadline` to be passed to `swapExactInput`).

It might be wise to add a deadline to the function parameters, to prevent of loss in huge changes in market conditions.

```
    function sellPoolTokens(
        uint256 poolTokenAmount,
```

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

[H-4] In `TSwapPool::_swap` the extra tokens given to users after `swapCount` reach 10, it 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 in `_swap` 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 until all the protocol funds are drained

Proof of Code

Place the following into `TSwapPool.t.sol`

```
    function testInvariantBroken() public {
```

```
        vm.startPrank(liquidityProvider);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.deposit(
            uint256(50e18),
            uint256(50e18),
            uint256(100e18),
            uint64(block.timestamp)
        );
        vm.stopPrank();

        uint256 outputWeth = 1e17;

        poolToken.mint(user, 100e18);
        weth.mint(user, 100e18);
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        weth.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();

        int256 endingY = int256(weth.balanceOf(address(pool)));
        int256 actualDeltaY = 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.

Remove these block from `_swap` function.

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

And also remove these lines from contract definition.

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

Medium

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

Description: The `deposit` function accepts a deadline parameter, which according to the 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.

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

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

- Found in TSwapPool.sol Line: 130

```
    uint64 deadline
```

Recommended Mitigation: Consider making the following change to the function.

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

```
10          returns (uint256 liquidityTokensToMint)
11      {
12          ...
13      }
```

[M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol's invariant

Description: The weird ERC20 tokens behave differently in contrary of normal ERC20s, the `TSwapPool` contract does not have any protection against these type of weird tokens. As a result when a liquidity provider funded, the pool will not get that much pool tokens the user determined and would have affect on the ratio from the start of the pool. And also it has the same impact when a user makes a swap between two tokens and the user will not get the correct amount.

Impact: fee-on-transfer affects on the constant product formula and change the ratio as expected to be in the first place before any pool tokens transfer.

Proof of Concept: Let's say USDC token will take %5 fee-on-transfer. The pool expects to get 100 tokens in exchange for WETH, but the USDC contract will take %5 fee and the pool will get only 95 tokens, but the evaluation of the exchange rate would 100 and the transaction will be taking place with 100 instead of the correct 95 tokens.

Add the following codes into the project

PoC

Create the following weird ERC20 fee-on-transfer contract in the path `test/mocks`, and named the file `TestWeirdERC20Mock.t.sol`.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.20;
3
4  import {Test, console} from "forge-std/Test.sol";
5  import {WeirdERC20Mock} from "./mocks/WeirdERC20Mock.sol";
6
7  contract TestWeirdERC20Mock is Test {
8      address owner = makeAddr("owner");
9      address sender = makeAddr("sender");
10     address receiver = makeAddr("receiver");
11
12     uint256 constant SEND_VALUE = 10e18;
13
14     WeirdERC20Mock weirdERC20;
15
16     constructor() {
17         vm.startPrank(owner);
18         weirdERC20 = new WeirdERC20Mock();
19         vm.stopPrank();
```

```
20
21          vm.startPrank(sender);
22          vm.deal(sender, 100e18);
23          weirdERC20.mint(sender, 100e18);
24          vm.stopPrank();
25      }
26
27      function testFeeOnTransfer() public {
28          vm.startPrank(sender);
29          weirdERC20.transfer(receiver, SEND_VALUE);
30          vm.stopPrank();
31
32          uint256 fee = weirdERC20.calculateFeeOnTransfer(SEND_VALUE);
33          uint256 amountReceived = SEND_VALUE - fee;
34
35          console.log(weirdERC20.balanceOf(owner));
36
37          assertEq(weirdERC20.balanceOf(receiver), amountReceived);
38      }
39  }
```

And include this block of code into `TSwapPool.t.sol`

```
1       function testBreakWithWeirdERC20FeeOnTransfer() public {
2           address owner = makeAddr("owner");
3           address provider = makeAddr("provider");
4           address swapper = makeAddr("swapper");
5
6           uint256 swap_value = 1e18;
7
8           WeirdERC20Mock weirdERC20;
9
10          vm.startPrank(owner);
11          weirdERC20 = new WeirdERC20Mock();
12          TSwapPool weirdPool = new TSwapPool(
13              address(weirdERC20),
14              address(weth),
15              "weird",
16              "WEIRD"
17          );
18          vm.stopPrank();
19
20          vm.startPrank(provider);
21          vm.deal(provider, 100e18);
22          weth.mint(provider, 100e18);
23          weirdERC20.mint(provider, 100e18);
24          weth.approve(address(weirdPool), type(uint256).max);
25          weirdERC20.approve(address(weirdPool), type(uint256).max);
26          weirdPool.deposit(100e18, 100e18, 100e18, uint64(block.
                timestamp));
27          vm.stopPrank();
```

```
28
29          uint256 weirdBalanceBeforeSwap = weirdERC20.balanceOf(address(
                weirdPool));
30          uint256 wethBalanceBeforeSwap = weth.balanceOf(address(
                weirdPool));
31          console.log(weirdBalanceBeforeSwap);
32
33          uint256 expectedChangeToPoolBalance = weirdPool.
                getOutputAmountBasedOnInput(swap_value,
                weirdBalanceBeforeSwap, wethBalanceBeforeSwap);
34
35          vm.startPrank(swapper);
36          weirdERC20.mint(swapper, 1e18);
37          weth.approve(address(weirdPool), type(uint256).max);
38          weirdERC20.approve(address(weirdPool), type(uint256).max);
39          weirdPool.swapExactInput(
40              weirdERC20,
41              swap_value,
42              weth,
43              9e17,
44              uint64(block.timestamp)
45          );
46          vm.stopPrank();
47
48          uint256 expectedBalanceAfterSwap = weirdBalanceBeforeSwap +
                expectedChangeToPoolBalance;
49          uint256 weirdBalanceAfterSwap = weirdERC20.balanceOf(address(
                weirdPool));
50          uint256 wethBalanceAfterSwap = weth.balanceOf(address(weirdPool
                ));
51          console.log(weirdBalanceAfterSwap);
52          console.log(wethBalanceAfterSwap);
53
54          assertEq(weirdBalanceAfterSwap, expectedBalanceAfterSwap);
55      }
```

Recommended Mitigation: As best practice it is better to monitor the pool reserve before and after any transactions if any huge unexpected changes happened could react to it.

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order causing event to emit incorrect information

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTrans` function, it logs values in an incorrect order. The `poolTokensToDeposit` should go in the third parameter position, whereas the `wethToDeposit` value should go second.

Impact: Event emission is incorrect, leading to off-chain functions potentially malfunctioning.

Recommended Mitigation: Make the changes as follows

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

[L-2] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given

Description: 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 is never assigned a value, nor uses an explicit return statement.

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

Proof of Concept: Add the following code to `TSwapPool.t.sol`

PoC

```
    function testOnIncorrectReturnedValue() public {
        uint256 wethLiquidity = 50e18;
        uint256 poolTokenLiquidity = 100e18;

        vm.startPrank(liquidityProvider);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.deposit(
            wethLiquidity,
            wethLiquidity,
            poolTokenLiquidity,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        uint256 expectedOutputAmount = pool.getOutputAmountBasedOnInput
            (10e18, poolToken.balanceOf(address(pool)), weth.balanceOf(
            address(pool)));

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        uint256 atcualOutputAmount = pool.swapExactInput(poolToken, 10
            e18, weth, 4e18, uint64(block.timestamp));
        vm.stopPrank();

        assertEq(atcualOutputAmount, expectedOutputAmount);
    }
```

Recommended Mitigation:

Changes

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (
-           uint256 output
+           uint256 outputAmount
        )
    {
        ...
        uint256 outputAmount = getOutputAmountBasedOnInput(
            inputAmount,
            inputReserves,
            outputReserves
        );
        ...
    }
```

Information

[I-1] Import `IERC20` interface from forge-std in `PoolFactory` may have some issues

Description: Import `IERC20` interface from `forge-std` library could include with some bugs and issues that not be reviewed and audited, so better to import it directly from OpenZeppelin contracts

Impact: May have some issue and the `PoolFactory` behave weirdly with ERC20s

Recommended Mitigation: Import `IERC20` directly from OpenZeppelin contracts

[I-2]: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all the fields should be indexed.

4 Found Instances

- Found in src/PoolFactory.sol Line: 38

```
1       event PoolCreated(address tokenAddress, address poolAddress);
```

- Found in src/TSwapPool.sol Line: 54

```
1       event LiquidityAdded(address indexed liquidityProvider,
            uint256 wethDeposited, uint256 poolTokensDeposited);
```

- Found in src/TSwapPool.sol Line: 59

```
1       event LiquidityRemoved(address indexed liquidityProvider,
            uint256 wethWithdrawn, uint256 poolTokensWithdrawn);
```

- Found in src/TSwapPool.sol Line: 64

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

[I-3] Lack of zero check on `wethToken` address parameter in `PoolFactory` constructor cause end up to set zero address

Description: There is no zero check in `PoolFactory` constructor to check `wethToken` parameter not be zero, it could set zero (burn) address as WETH token address

Impact: Could make whole `PoolFactory` contract useless and to fix this have to redeploy it

Recommended Mitigation: Change the constructor as follows

Code

```
1       constructor(address wethToken) {
2   +       if (wethToken != address(0)) {
3               i_wethToken = wethToken;
4   +       }
5       }
```

[I-4] Mistakenly fetching the name of the pool token instead of its symbol

Description: Fetching the name of the pool token for creating the pool instead of its symbol to put it as the pool ERC20 symbol name, might usage gas of this process goes up cause the name variable usually has more value in it

Impact: The name variable would be longer that symbol, and might gas usage goes up a little

Recommended Mitigation: Correct the `createPool` function as follows

Code

```
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).symbol());
-       string memory liquidityTokenSymbol = string.concat("ts", IERC20
    (tokenAddress).name());
        TSwapPool tPool = new TSwapPool(
            tokenAddress,
            i_wethToken,
            liquidityTokenName,
            liquidityTokenSymbol
        );
        s_pools[tokenAddress] = address(tPool);
        s_tokens[address(tPool)] = tokenAddress;
        emit PoolCreated(tokenAddress, address(tPool));
        return address(tPool);
    }
```

[I-5] `PoolFactory__PoolDoesNotExist` error never use in the contract

Description: The `PoolFactory__PoolDoesNotExist` error never use in the contract and the meaning of this error not quite reasonable, so it's better to remove it from the contract

Impact: Will more gas used while deploying a bit and the complexity of the code be higher

Recommended Mitigation: Delete the `PoolFactory__PoolDoesNotExist` error from the codebase

[I-6] `MINUMUM_WETH_LIQUIDITY` is a constant variable and does not need to be emitted from `TSwapPool__WethDepositAmountTooLow` event

Description: The `MINUMUM_WETH_LIQUIDITY` is a constant variable storage state and does not need to be emitted from `TSwapPool__WethDepositAmountTooLow` event in `TSwapPool::deposit` function cause it already accessible in codebase with ease, and it's kind of waste of gas only

Recommended Mitigation: Remove `minimumWethDeposit` from `TSwapPool__WethDepositAmountTooLow` event

Code

```
error TSwapPool__WethDepositAmountTooLow(
-   uint256 minimumWethDeposit,
    uint256 wethToDeposit
);
```

[I-7] It is better to set `liquidityTokensToMint` before external calls in `TSwapPool::deposit` function

Description: Should be better to set `liquidityTokensToMint` before swap calls in `TSwapPool::deposit` function in case no liquidity already be in the pool because the call is external and better to follow CEI and impacts on contract states before external calls happen

Recommended Mitigation: In case no liquidity in the pool we change the code like so

Code

```
if {
    ...
} else {
    // This will be the "initial" funding of the protocol. We are
       starting from blank here!
    // We just have them send the tokens in, and we mint liquidity
       tokens based on the weth
+   liquidityTokensToMint = wethToDeposit;
    _addLiquidityMintAndTransfer(
        wethToDeposit,
        maximumPoolTokensToDeposit,
        wethToDeposit
    );
-   liquidityTokensToMint = wethToDeposit;
}
```

[I-8] Lack of zero check in `TSwapPool` constructor, cause to set zero address on WETH and pool token, would break contract

Description: The lack of zero check in `TSwapPool` constructor would cause to deploy a break contract with no functionality and waste all the gas for deploying wrong token addresses

Impact: Set zero (burn) addresses on WETH and Pool token and would be a broke contract, and also be wasting of gas

Recommended Mitigation: Change the constructor as follows:

Code

```
constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
+       if (poolToken != address(0) && wethToken != address(0)) {
            i_wethToken = IERC20(wethToken);
            i_poolToken = IERC20(poolToken);
+       }
    }
```

[I-9] Missing `revertIfZero` modifier check on `inputReserves` in `TSwapPool::getOutputAmountBasedOnInput`

Description: Missing `revertIfZero` modifier check on `inputReserves` in `TSwapPool::getOutputAmountBasedOnInput` could encounter to wrong calculation of the output token.

Impact: Would encounter to wrong calculation of output token. Although we get the reserve in callers other function but as best practice we should check this parameter since it is a public function

Recommended Mitigation: Add this line to `TSwapPool::getOutputAmountBasedOnInput` function definition

Code

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

[I-10] Magic numbers in `TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool::getInputAmountBasedOnOutput` functions, might boil down to set wrong numbers in the code

Description: Magic numbers in `TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool::getInputAmountBasedOnOutput` functions, might boil down to set wrong numbers since these are important functions to get correct amount to swap. Changing them to constant variables could help of readability of the code.

Impact: The literal numbers could set mistakenly and interruption to the functionality of the code.

Recommended Mitigation: Create two constant variables and replace them instead of literal number in `getOutputAmountBasedOnInput` function.

[I-11] Lack of inline documentation (NatSpec) in important functions

Description: The lack of inline documentation (natspec) make it hard to understand of the functionality of the functions and their purposes. Functions don't have natspec such as `getOutputAmountBasedOnInput`, `getInputAmountBasedOnOutput`, `swapExactInput`, `swapExactOutput` and `getPoolTokensToDepositBasedOnWeth`.

Impact: Could make it hard to understand the functionality of the function and might a function do something that not mentioned at all and the auditors would get it wrong

Recommended Mitigation: Write appropriate inline documentations specially for important functions that perform swap and change the states of the contract

[I-12] Missing `revertIfZero` modifier check on `inputReserves` in `TSwapPool::getInputAmountBasedOnOutput`

Description: Missing `revertIfZero` modifier check on `inputReserves` in `TSwapPool::getInputAmountBasedOnOutput` could encounter to wrong calculation of the output token.

Impact: Would encounter to wrong calculation of output token. Although we get the reserve in callers other function but as best practice we should check this parameter since it is a public function

Recommended Mitigation: Add this line to `TSwapPool::getInputAmountBasedOnOutput` function definition

Code

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

[I-13] Using literal numbers in `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` functions, magic numbers could be lead to make a mistake

Description: Using literal numbers in `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` functions, magic numbers could lead to make a mistake and should not as so, instead declare constant variable or a normal one to call them directly and maintain conveniently.

Impact: Could be lead to make a mistake to put numbers in the code directly.

Recommended Mitigation: Add a new constant variable in contracts definition

Code

```
contract TSwapPool is ERC20 {
...
+   uint256 constant public ONE_TOKEN = 1e18;
...
}
```

And change the code in two functions as follows

Code

```
    function getPriceOfOneWethInPoolTokens() external view returns (
       uint256) {
        return
            getOutputAmountBasedOnInput(
+               ONE_TOKEN,
-               1e18,
                i_wethToken.balanceOf(address(this)),
```

```
7                 i_poolToken.balanceOf(address(this))
8             );
9     }
10
11    function getPriceOfOnePoolTokenInWeth() external view returns (
          uint256) {
12        return
13            getOutputAmountBasedOnInput(
14 +              ONE_TOKEN,
15 -              1e18,
16                i_poolToken.balanceOf(address(this)),
17                i_wethToken.balanceOf(address(this))
18            );
19    }
```

Gas

[G-1] `poolTokenReserves` never used in the `TSwapPool::deposit` function and will waste gas

Description: The `poolTokenReserves` never used in the `TSwapPool::deposit` function and would be just wasting of gas

Impact: Wasting of gas to define and set `poolTokenReserves` in the function

Recommended Mitigation: Delete this line of code in deposit function

Code

```
1  function deposit(
2          uint256 wethToDeposit,
3          uint256 minimumLiquidityTokensToMint,
4          uint256 maximumPoolTokensToDeposit,
5          uint64 deadline
6      )
7          external
8          revertIfZero(wethToDeposit)
9          returns (uint256 liquidityTokensToMint)
10     {
11         ...
12         if (totalLiquidityTokenSupply() > 0) {
13             uint256 wethReserves = i_wethToken.balanceOf(address(this))
                   ;
14 -           uint256 poolTokenReserves = i_poolToken.balanceOf(address(
       this));
15             // Our invariant says weth, poolTokens, and liquidity
                   tokens must always have the same ratio after the
16             // initial deposit
```

```
17              // poolTokens / constant(k) = weth
18              ...
19      }
```

[G-2] The `TSwapPool::swapExactInput` should declare as external it would use more gas

Description: The `TSwapPool::swapExactInput` should declare as external since it is not called internally and public function would use more gas usually.

Impact: The usage of gas would be higher in public functions.

Proof of Concept:

Recommended Mitigation: Define the `swapExactInput` function's visibility to external