TSwap
Protocol Audit Report

Version 1.2

Oleh Yatskiv at OmiSoft

May 21, 2024

TSwap Protocol Audit Report

Oleh Yatskiv at OmiSoft

May 21st, 2024

Prepared by: OmiSoft team

Lead Auditor: Oleh Yatskiv

Table of Contents

    * [H-3] The `TSwapPool::swapExactOutput` function is missing slippage protection, which can lead to a loss of funds for the user during the swap

- Medium
    * [M-1] Function `TSwapPool::sellPoolTokens` calls the wrong `swap` function, leading to potential fund losses for the user
    * [M-2] The `deadline` variable in `TSwapPool::deposit` function is ignored, leading to unexpected behavior from the function as user expects the `deadline` to be taken into account
    * [M-3] Rebase, fee-on-transfer and ERC-777 tokens break the protocol invariant
- Low
    * [L-1] The `TSwapPool::getInputAmountBasedOnOutput` function lacks some necessary checks, resulting in overflows and division by zero
    * [L-2] `TSwapPool::_addLiquidityMintAndTransfer` has swapped parameters for an event emission, which can led to confusion and wrong data interpretation
    * [L-3] The `TSwapPool::swapExactInput` has an unused function return parameter `output` that always defaults to 0, which can be misleading for the users
- Gas
    * [G-1] Unused error in `PoolFactory` contract could be removed
    * [G-2] Excessive usage of a modifier in the `TSwapPool::deposit` function
    * [G-3] Unused local variable in the `TSwapPool::deposit` function could be removed
    * [G-4] All **`public`** functions not used internally could be marked `external`
    * [G-5] The `TSwapPool::totalLiquidityTokenSupply` function should either be marked as `external` and not used internally, or removed altogether
- Informational
    * [I-1] PUSH0 is not supported by all chains
    * [I-2] Events are missing `indexed` fields
    * [I-3] The `PoolFactory` and `TSwapPool` contracts are missing natspec comments for the contracts and some of the functions
    * [I-4] The `PoolFactory` and `TSwapPool` contracts' elements does not follow the Solidity style guide order of layout
    * [I-5] Zero checks for addresses in the `PoolFactory` and `TSwapPool` contracts constructors are missing
    * [I-6] Wrong function call at `PoolFactory::createPool` resulting in a wrong LP token symbol
    * [I-7] The `TSwapPool::TSwapPool__WethDepositAmountTooLow` error's variable `minimumWethDeposit` is actually a constant, and should not be emmitted

* [I-8] Large literal values multiples of 10 can be replaced with scientific notation
* [I-9] Misleading or confusing error messages
* [I-10] Calculation logic could be moved to another function
* [I-11] The `TSwapPool::deposit` function does not pollow the `CEI` pattern
* [I-12] Misleading natspec comment in `@dev` section of the `TSwapPool::_addLiquidityMintAndTransfer` function
* [I-13] Usage of `magic numbers` is not advisable
* [I-14] The `TSwapPool::sellPoolTokens` function should be removed

Protocol Summary

T-swap is an automated market maker (AMM) that allows users to swap tokens. It consists of two main contracts: `PoolFactory` and `TSwapPool`. The `PoolFactory` contract is used to create new pools of tokens and supply them, while the `TSwapPool` contract is used to swap tokens between exactly two assets mainly utilizing the `swapExactInput` and `swapExactOutput` functions. The protocol starts as a `PoolFactory` contract, which is used to create new pools of tokens.

Disclaimer

Oleh Yatskiv from the OmiSoft 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

The findings described in this document correspond to 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 issues found
High	3
Medium	3
Low	3
Gas	5
Info	14
Total	28

Findings

High

[H-1] The incentive logic in the `TSwapPool::_swap` function leads to an invariant break

Description: From the natspec comment on the `TSwapPool::_swap` function: `Every 10 swaps, we give the caller an extra token as an extra incentive to keep trading on T-Swap`. The function transfers `1e18` of output tokens to the caller every 10 swaps, which breaks the core invariant of the protocol that the pool should always have the same ratio of WETH and pool tokens.

```
    function _swap(IERC20 inputToken, uint256 inputAmount, IERC20
        outputToken, uint256 outputAmount) private {
        ...
        swap_count++;
        if (swap_count >= SWAP_COUNT_MAX) {
            swap_count = 0;
            outputToken.safeTransfer(msg.sender, 1
                _000_000_000_000_000_000);
        }
        ...
    }
```

Impact: This incentive logic can lead to the pool running out of WETH or pool tokens, breaking the invariant and severely disrupting the protocol's functionality. An attacker could also just drain the pool by making small transfers.

Proof of Concept: The following test function does 10 swaps, and shows that the pool reserves are now imbalanced, and the core invariant is broken:

```
    function testBreaksTheInvariant() public {
        uint256 inputReserves = 200e18;
        uint256 outputReserves = 200e18;
        uint256 EXCLUDE_FEE_NUMERATOR = 997;
        uint256 EXCLUDE_FEE_DENOMINATOR = 1000;
        uint256 poolReservesInvariantStart = inputReserves *
            outputReserves;
        console.log("Pool reserves invariant start: ",
            poolReservesInvariantStart);
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), inputReserves);
        poolToken.approve(address(pool), outputReserves);
        pool.deposit(inputReserves, outputReserves, outputReserves,
            uint64(block.timestamp));
        vm.stopPrank();

```

```
14          vm.startPrank(user);
15          for (uint256 i = 0; i < 9; i++) {
16              i % 2 == 0 ? poolToken.approve(address(pool), poolToken.
                    balanceOf(user))
17                         : weth.approve(address(pool), weth.balanceOf(
                               user));
18              i % 2 == 0 ? pool.swapExactInput(poolToken, poolToken.
                    balanceOf(user), weth, 0, uint64(block.timestamp))
19                         : pool.swapExactInput(weth, weth.balanceOf(user)
                               , poolToken, 0, uint64(block.timestamp));
20              uint256 poolReservesInvariantInBetween = weth.balanceOf(
                    address(pool)) * poolToken.balanceOf(address(pool));
21              console.log("Pool reserves invariant in-between (we
                    accumulate fees): ", poolReservesInvariantInBetween);
22          }
23
24          int256 expectedDeltaY = int256(-1) * int256(weth.balanceOf(user
                ) * EXCLUDE_FEE_NUMERATOR / EXCLUDE_FEE_DENOMINATOR);
25          int256 startingY = int256(poolToken.balanceOf(address(pool)));
26          weth.approve(address(pool), weth.balanceOf(user));
27          pool.swapExactInput(weth, weth.balanceOf(user), poolToken, 0,
                uint64(block.timestamp));
28          int256 endingY = int256(poolToken.balanceOf(address(pool)));
29          int256 actualDeltaY = endingY - startingY;
30          uint256 poolReservesInvariantEnd = weth.balanceOf(address(pool)
                ) * poolToken.balanceOf(address(pool));
31          console.log("Pool reserves invariant end: ",
                poolReservesInvariantEnd);
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

The assertion fails, indicating that the pool lost from the swap more than intended:

$$[FAIL.\ Reason:\ assertion\ failed:\ -2002357781824619991 != -1901839372463681936 7]$$

We can also see the invariant change throughout the swaps in the logs. It should only increase (accumulating fees), but we see that it decreases at the end by the 1e18 amount we sent to the user:

```
1  Logs:
2    Pool reserves invariant in-between:
         400057151021574510645300000000000000000000
3    Pool reserves invariant in-between:
         400168607095358266701900000000000000000000
4    Pool reserves invariant in-between:
         400279790642320881255600000000000000000000
5    Pool reserves invariant in-between:
         400390702160802895128300000000000000000000
6    Pool reserves invariant in-between:
         400501342149158904511200000000000000000000
7    Pool reserves invariant in-between:
         400611711105751501981800000000000000000000
8    Pool reserves invariant in-between:
```

```
        400721809528945250700900000000000000000000
9     Pool reserves invariant in-between:
        400831637917100691739200000000000000000000
10    Pool reserves invariant in-between:
        400941196768568384481000000000000000000000
11    Pool reserves invariant start:
        400000000000000000000000000000000000000000
12    Pool reserves invariant end:
        398950486581682980018900000000000000000000
```

Recommended Mitigation: It is recommended to remove the incentive logic from the `_swap` function, as it breaks the core invariant of the protocol. The incentive logic can be moved to another function that would not affect the pool reserves, and will use a different fund to keep the incentive program running.

[H-2] `TSwapPool::getInputAmountBasedOnOutput` uses wrong denominator coefficient, leading to a huge fee percentage being taken from the user during the swap

Description: The `TSwapPool::getInputAmountBasedOnOutput` function uses the wrong denominator coefficient, leading to `90.03%` fees being taken from the user during the `swapExactOutput` swap instead of `0.3%`. The function should use the `1000` denominator coefficient, but it uses the `10000` denominator coefficient instead:

```
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) * 10000) / ((
       outputReserves - outputAmount) * 997);
13     }
```

Impact: The user expects to pay `0.3%` fees during the swap, but instead, he pays `90.03%` fees, which is a significant difference. This severely disrupts the protocol's functionality and leads to a loss of funds for the user during the `swapExactOutput` swap.

Proof of Concept: We can easily check that by calling `swapExactOutput` in our function and checking the amount of Pool Token we need to spend to get the desired amount of WETH. Then we compare it to the expected input:

```
    function testWrongFees() public {
        uint256 inputReserves = 200e18;
        uint256 outputReserves = 200e18;
        uint256 amountIn = 100e18;
        uint256 WITH_FEE_NUMERATOR = 1003;
        uint256 WITH_FEE_DENOMINATOR = 1000;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), inputReserves);
        poolToken.approve(address(pool), outputReserves);
        pool.deposit(inputReserves, outputReserves, outputReserves,
            uint64(block.timestamp));
        vm.stopPrank();
        vm.startPrank(user);
        poolToken.mint(user, amountIn * 1e4);
        poolToken.approve(address(pool), amountIn * 1e4);
        uint256 output = amountIn;
        uint256 input = pool.swapExactOutput(poolToken, weth, output,
            uint64(block.timestamp));
        uint256 expectedInput = amountIn * WITH_FEE_NUMERATOR /
            WITH_FEE_DENOMINATOR;
        console.log("Input: ", input);
        console.log("Expected input: ", expectedInput);
        assertEq(input, expectedInput);
    }
```

The assertion fails, indicating that the user pays more than expected. We can see from the logs, that user has to pay 20 times more than expected:

```
Logs:
  Input:  2006018054162487462387
  Expected input:  10030000000000000000
```

Recommended Mitigation: It is recommended to use the correct denominator coefficient in the `TSwapPool::getInputAmountBasedOnOutput` function to prevent the user from paying more fees than expected. The function should use the `1000` denominator coefficient instead of the `10000` denominator coefficient. It would also be best to just use constants for the fee calculation as suggested in the `[I-13]` issue.

[H-3] The `TSwapPool::swapExactOutput` function is missing slippage protection, which can lead to a loss of funds for the user during the swap

Description: The `TSwapPool::swapExactOutput` function is missing slippage protection, which can lead to a loss of funds for the user during the `swapExactOutput` swap. The user expects the swap to revert if the slippage is too high, but the function does not take the `maximumInputAmount` parameter as it should, leading to inability for the user to set his desired maximum of input tokens

used.

```
    function swapExactOutput(
        IERC20 inputToken,
->
        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);

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

Impact: The user expects the swap to revert if the slippage is too high, but the function does not take the `maximumInputAmount` parameter as it should, leading to a loss of funds for the user during the `swapExactOutput` swap if the market price moves too much.

Proof of Concept: We can even take the PoC from the `[H-1]` issue, as it perfectly shows us the inability to set the desired maximum of input tokens used. That way the function uses way more of the tokens that user intended to use, leading to a loss of funds for the user:

```
    function testNoSlippageProtection() public {
        uint256 inputReserves = 200e18;
        uint256 outputReserves = 200e18;
        uint256 amountIn = 100e18;
        uint256 WITH_FEE_AND_SLIPPAGE_NUMERATOR = 1013; // We are
            willing to have a maximum of 1% slippage (the fee is 0.3%)
        uint256 WITH_FEE_AND_SLIPPAGE_DENOMINATOR = 1000;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), inputReserves);
        poolToken.approve(address(pool), outputReserves);
        pool.deposit(inputReserves, outputReserves, outputReserves,
            uint64(block.timestamp));
        vm.stopPrank();
        vm.startPrank(user);
        poolToken.mint(user, amountIn * 1e4);
        poolToken.approve(address(pool), amountIn * 1e4);
        uint256 output = amountIn;
        uint256 input = pool.swapExactOutput(poolToken, weth, output,
            uint64(block.timestamp));
```

```
17          uint256 expectedMaximumInput = amountIn *
                WITH_FEE_AND_SLIPPAGE_NUMERATOR /
                WITH_FEE_AND_SLIPPAGE_DENOMINATOR;
18          console.log("Input: ", input);
19          console.log("Expected maximum input: ", expectedMaximumInput);
20          assertGe(expectedMaximumInput, input);
21      }
```

Assertion fails, indicating that the user pays more than expected. We can see from the logs, that user has to pay 19.8 times more than expected:

```
1  Logs:
2    Input:  2006018054162487462387
3    Expected maximum input:  101300000000000000000
```

Recommended Mitigation: It is recommended to add slippage protection to the `TSwapPool::swapExactOutput` function to prevent a loss of funds for the user during the swap. The function should take the `maximumInputAmount` parameter, and revert if the slippage is too high. This can be done by adding the `maximumInputAmount` parameter to the function and checking if the `inputAmount` is less than or equal to the `maximumInputAmount`:

```
1       function swapExactOutput(
2           IERC20 inputToken,
3   ->      uint256 maximumInputAmount,
4           IERC20 outputToken,
5           uint256 outputAmount,
6           uint64 deadline
7       )
8           public
9           revertIfZero(outputAmount)
10          revertIfDeadlinePassed(deadline)
11          returns (uint256 inputAmount)
12      {
13          uint256 inputReserves = inputToken.balanceOf(address(this));
14          uint256 outputReserves = outputToken.balanceOf(address(this));
15
16          inputAmount = getInputAmountBasedOnOutput(outputAmount,
                inputReserves, outputReserves);
17  ->      if (inputAmount > maximumInputAmount) {
18  ->          revert TSwapPool__InputHigherThanExpectedMax(inputAmount,
        maximumInputAmount);
19  ->      }
20
21          _swap(inputToken, inputAmount, outputToken, outputAmount);
22      }
```

Medium

[M-1] Function `TSwapPool::sellPoolTokens` calls the wrong swap function, leading to potential fund losses for the user

Description: The `TSwapPool::sellPoolTokens` function calls `swapExactOutput` instead of `swapExactInput` (as we are given the exact input amount of Pool Tokens to sell). The function will try to sell the exact amount of Pool Tokens as the exact amount of WETH, leading to a potential unwanted swap or reverts:

```
1      function sellPoolTokens(uint256 poolTokenAmount) external returns (
           uint256 wethAmount) {
2  ->      return swapExactOutput(i_poolToken, i_wethToken,
       poolTokenAmount, uint64(block.timestamp));
3      }
```

Impact: The user expects to sell the exact amount of Pool Tokens for WETH, but the function will try to sell the amount of Pool Tokens as the exact amount of WETH, leading to unwanted swaps or reverts, which can lead to potential fund losses.

Proof of Concept: We can write a simple test function that tries to sell the exact amount of Pool Tokens (our whole balance) for WETH, and check if the function reverts or not:

```
1      function testSellPoolTokens() public {
2          uint256 inputReserves = 200e18;
3          uint256 outputReserves = 200e18;
4          uint256 amountIn = 100e18;
5          vm.startPrank(liquidityProvider);
6          weth.approve(address(pool), inputReserves);
7          poolToken.approve(address(pool), outputReserves);
8          pool.deposit(inputReserves, outputReserves, outputReserves,
               uint64(block.timestamp));
9          vm.stopPrank();
10         vm.startPrank(user);
11         poolToken.mint(user, amountIn);
12         poolToken.approve(address(pool), amountIn);
13         vm.expectRevert();
14         pool.sellPoolTokens(amountIn);
15     }
```

The test passes, meaning that the function reverted, i.e. the user can't sell the amount of tokens he intended to sell.

Recommended Mitigation: As stated in the `[I-14]` issue, it is best to just remove the `swapExactOutput` function altogether. However, if we want to keep it, we should change the function to call the `swapExactInput` function instead of the `swapExactOutput` function, as

we are given the exact input amount of Pool Tokens to sell. We also want to take into account the `deadline`, and the `minimumExpectedOutput`:

```
    function sellPoolTokens(uint256 poolTokenAmount, uint256
        minimumExpectedOutput, uint64 deadline) external returns (
        uint256 wethAmount) {
->      return swapExactInput(i_poolToken, poolTokenAmount, i_wethToken
    , minimumExpectedOutput, deadline);
    }
```

[M-2] The `deadline` variable in `TSwapPool::deposit` function is ignored, leading to unexpected behavior from the function as user expects the `deadline` to be taken into account

Description: The `TSwapPool::deposit` function takes a `deadline` variable as an argument, but it is not used in the function. The user expects the `deadline` to be taken into account when depositing WETH and pool tokens into the pool, but the function does not use it, leading to unexpected behavior from the function:

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
->      uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        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 poolTokensToDeposit =
                getPoolTokensToDepositBasedOnWeth(wethToDeposit);
            if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
                revert TSwapPool__MaxPoolTokenDepositTooHigh(
                    maximumPoolTokensToDeposit, poolTokensToDeposit);
            }
            liquidityTokensToMint = wethToDeposit *
                totalLiquidityTokenSupply() / wethReserves;
            if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
                revert TSwapPool__MinLiquidityTokensToMintTooLow(
                    minimumLiquidityTokensToMint, liquidityTokensToMint)
```

```
                        ;
24              }
25              _addLiquidityMintAndTransfer(wethToDeposit,
                    poolTokensToDeposit, liquidityTokensToMint);
26          } else {
27              _addLiquidityMintAndTransfer(wethToDeposit,
                    maximumPoolTokensToDeposit, wethToDeposit);
28              liquidityTokensToMint = wethToDeposit;
29          }
30      }
```

Impact: All of the transactions that should have been expired and expected to be expired by the user will go through anyway, leading to frustration, and potentially loss of funds for the user (he might try depositing again, when he expects the function to fail, but would actually just deposit twice).

Proof of Concept: We can write a simple function that deposits the funds to the pool with an expired `deadline` timestamp. The function should revert, as the `deadline` is expired:

```
1       function testDeadlineNotTakenIntoAccount() public {
2           uint256 inputReserves = 100e18;
3           uint256 outputReserves = 100e18;
4           vm.startPrank(liquidityProvider);
5           weth.approve(address(pool), inputReserves);
6           poolToken.approve(address(pool), outputReserves);
7           uint64 expiredDeadline = uint64(block.timestamp - 1);
8           vm.expectRevert();
9           pool.deposit(inputReserves, outputReserves, outputReserves,
                expiredDeadline);
10      }
```

However, the function does not revert, meaning that the `deadline` is not taken into account: $[FAIL.\ Reason:\ call\ did\ not\ revert\ as\ expected]$

Recommended Mitigation: It is recommended to use the `deadline` variable in the `TSwapPool::deposit` function to prevent unexpected behavior from the function. The function should revert if the `deadline` is expired. This can be done by utilizing the existing modifier `revertIfDeadlinePassed()`:

```
1       function deposit(
2           uint256 wethToDeposit,
3           uint256 minimumLiquidityTokensToMint,
4           uint256 maximumPoolTokensToDeposit,
5           uint64 deadline
6       )
7           external
8           revertIfZero(wethToDeposit)
9   ->      revertIfDeadlinePassed(deadline)
10          returns (uint256 liquidityTokensToMint)
11      {
```

```
12          ...
13      }
```

[M-3] Rebase, fee-on-transfer and ERC-777 tokens break the protocol invariant

Description: Different types of non-standard ERC-20 tokens with additional functionality might easily break the protocol invariant. For example, rebasing tokens, fee-on-transfer tokens, and ERC-777 tokens can break the protocol invariant, as they can change the amount of tokens in the pool without the pool being aware of it. The pool should always have the same ratio of WETH and pool tokens, but these tokens can change the amount of tokens in the pool, leading to a break of the core invariant of the protocol.

Impact: The protocol invariant is broken, leading to a loss of funds for the users, and severely disrupting the protocol's functionality.

Proof of Concept: We can write a simple ERC-20 that would revert on each transfer:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.20;
3
4  import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
5
6  contract MaliciousERC20 is ERC20 {
7      constructor() ERC20("Malicious", "HACK") { }
8
9      function mint(address to, uint256 amount) public {
10         _mint(to, amount);
11     }
12
13     function transferFrom(address from, address to, uint256 value)
           public override returns (bool) {
14         revert("MaliciousERC20: transfer failed");
15     }
16 }
```

Then we can write a simple test function that tries to deposit the funds to the pool with the `MaliciousERC20` token:

```
1      function testNonStandardERC20transfer() public {
2          MaliciousERC20 maliciousToken = new MaliciousERC20();
3          weth = new ERC20Mock();
4          pool = new TSwapPool(address(maliciousToken), address(weth), "
               LTokenB", "LB");
5
6          weth.mint(liquidityProvider, 200e18);
7          maliciousToken.mint(liquidityProvider, 200e18);
8
```

```
        vm.startPrank(liquidityProvider);
        maliciousToken.approve(address(pool), 200e18);
        weth.approve(address(pool), 200e18);
        vm.expectRevert("MaliciousERC20: transfer failed");
        pool.deposit(200e18, 200e18, 200e18, uint64(block.timestamp));
    }
```

It would revert as expected, meaning that the pool invariant could easily be broken if there are any logic in the non-standard ERC-20 token that would temper the amount of tokens in the transfer logic.

Recommended Mitigation: After each operation that involves transferring tokens, the pool should check if the invariant is still valid. If it is not, the pool should revert the operation. It is also recommended to add a check for the ERC-777 tokens, as they can break the protocol invariant as well.

Low

[L-1] The `TSwapPool::getInputAmountBasedOnOutput` function lacks some necessary checks, resulting in overflows and division by zero

Description: The `TSwapPool::getInputAmountBasedOnOutput` function does not check that the `outputAmount` is bigger than `outputReserves` before doing the subtraction, which can lead to overflows and division by zero. This would be especially the case in small pools, where the `outputReserves` could be easily smaller than the desired `outputAmount`.

Impact: This can lead to overflows and division by zero, which can result in the contract reverting without giving any useful information to the user about the revert. It should clearly state that the output amount is too high for the reserves.

Proof of Concept: We can write a simple test function to see it in action. It will expect reverting with the "division by zero (0x12)" error when the `outputAmount` is equal to `outputReserver`, and reverting with the "overflow (0x11)" error when the `outputAmount` is bigger than `outputReserves`:

```
    function testDivisionByZeroAndOverflowReverts() public {
        uint256 inputReserves = 100e18;
        uint256 outputReserves = 100e18;
        uint256 amountOutForDivByZero = 100e18;
        uint256 amountOutForOverflow = 200e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), inputReserves);
        poolToken.approve(address(pool), outputReserves);
        pool.deposit(inputReserves, outputReserves, outputReserves,
            uint64(block.timestamp));

        vm.startPrank(user);
        poolToken.mint(user, amountOutForOverflow);
```

```
13          poolToken.approve(address(pool), amountOutForOverflow);
14          uint256 divisionByZeroCode = uint256(0x12); // 0x12 is division
                by zero
15          vm.expectRevert(abi.encodePacked(bytes4(keccak256("Panic(
                uint256)")), divisionByZeroCode));
16          pool.swapExactOutput(poolToken, weth, amountOutForDivByZero,
                uint64(block.timestamp));
17          uint256 overflowCode = uint256(0x11); // 0x11 is overflow
18          vm.expectRevert(abi.encodePacked(bytes4(keccak256("Panic(
                uint256)")), overflowCode));
19          pool.swapExactOutput(poolToken, weth, amountOutForOverflow,
                uint64(block.timestamp));
20      }
```

The test succeeds, meaning that we get the expected revert errors with division by zero and overflow.

Recommended Mitigation: We should add a check that the `outputAmount` is smaller than `outputReserves` before doing the subtraction. It would look something like this

```
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          if (outputAmount >= outputReserves) {
13              revert TSwapPool__OutputReservesTooLow(outputReserves,
                    outputAmount);
14          }
15          return ((inputReserves * outputAmount) * 1000) / ((
                outputReserves - outputAmount) * 997);
16      }
```

And also add a custom error `TSwapPool__OutputReservesTooLow` to give a more descriptive error message to the user:

```
1       error TSwapPool__OutputReservesTooLow(uint256 outputReserves,
            uint256 outputAmount);
```

[L-2] `TSwapPool::_addLiquidityMintAndTransfer` has swapped parameters for an event emission, which can led to confusion and wrong data interpretation

Description: The event `LiquidityAdded` is emitted in the `TSwapPool::_addLiquidityMintAndTransfer` function with the `poolTokensToDeposit` and `wethToDeposit` parameters swapped. Declaration:

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

Usage in the function:

```
1     function _addLiquidityMintAndTransfer(
2         uint256 wethToDeposit,
3         uint256 poolTokensToDeposit,
4         uint256 liquidityTokensToMint
5     )
6         private
7     {
8         ...
9 ->      emit LiquidityAdded(msg.sender, poolTokensToDeposit,
      wethToDeposit);
10        ...
11    }
```

Impact: This can lead to confusion and wrong data interpretation when listening to the event.

Recommended Mitigation: Swap the parameters in the event emission:

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

[L-3] The `TSwapPool::swapExactInput` has an unused function return parameter `output` that always defaults to 0, which can be misleading for the users

Description: The `TSwapPool::swapExactInput` function has an unused function return parameter `output` that always defaults to 0, which can be misleading for the users. The function should return the actual output amount of the swap, but it isn't set anywhere:

```
1     function swapExactInput(
2         IERC20 inputToken,
3         uint256 inputAmount,
4         IERC20 outputToken,
5         uint256 minOutputAmount,
6         uint64 deadline
7     )
8         public
```

```
9           revertIfZero(inputAmount)
10          revertIfDeadlinePassed(deadline)
11  ->      returns (uint256 output)
12      {
13          uint256 inputReserves = inputToken.balanceOf(address(this));
14          uint256 outputReserves = outputToken.balanceOf(address(this));
15          uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
                inputReserves, outputReserves);
16          if (outputAmount < minOutputAmount) {
17              revert TSwapPool__OutputTooLow(outputAmount,
                    minOutputAmount);
18          }
19          _swap(inputToken, inputAmount, outputToken, outputAmount);
20      }
```

Impact: This can be misleading for the users, as they expect the function to return the actual output amount of the swap, but it always defaults to 0.

Proof of Concept: We can write a simple test function that calls the `swapExactInput` function and checks the return value:

```
1       function testSwapExactInputReturnsZero() public {
2           uint256 inputReserves = 200e18;
3           uint256 outputReserves = 200e18;
4           uint256 amountIn = 100e18;
5           vm.startPrank(liquidityProvider);
6           weth.approve(address(pool), inputReserves);
7           poolToken.approve(address(pool), outputReserves);
8           pool.deposit(inputReserves, outputReserves, outputReserves,
                uint64(block.timestamp));
9           vm.stopPrank();
10          vm.startPrank(user);
11          poolToken.mint(user, amountIn);
12          poolToken.approve(address(pool), amountIn);
13          uint256 actualOutput = pool.swapExactInput(poolToken, amountIn,
                weth, 0, uint64(block.timestamp));
14          assertEq(actualOutput, 0);
15      }
```

The test passes, meaning that the function returns 0, which can be misleading for the users.

Recommended Mitigation: It is recommended to remove the `output` function return parameter from the `TSwapPool::swapExactInput` function, as it is not used and can be misleading for the users. You can also return the actual output amount of the swap:

```
1       function swapExactInput(
2           IERC20 inputToken,
3           uint256 inputAmount,
4           IERC20 outputToken,
5           uint256 minOutputAmount,
```

```
6           uint64 deadline
7       )
8           public
9           revertIfZero(inputAmount)
10          revertIfDeadlinePassed(deadline)
11          returns (uint256 outputAmount)
12      {
13          uint256 inputReserves = inputToken.balanceOf(address(this));
14          uint256 outputReserves = outputToken.balanceOf(address(this));
15  ->      outputAmount = getOutputAmountBasedOnInput(inputAmount,
        inputReserves, outputReserves);
16          if (outputAmount < minOutputAmount) {
17              revert TSwapPool__OutputTooLow(outputAmount,
                    minOutputAmount);
18          }
19          _swap(inputToken, inputAmount, outputToken, outputAmount);
20      }
```

Gas

[G-1] Unused error in `PoolFactory` contract could be removed

The `PoolFactory` contract defines an error `PoolFactory__PoolDoesNotExist` that is not used in the contract. It is recommended to remove the definition of the error to keep the code clean and save gas costs.

[G-2] Excessive usage of a modifier in the `TSwapPool::deposit` function

We use the `revertIfZero(wethToDeposit)` modifier in the `TSwapPool::deposit` function to check if the `wethToDeposit` variable is not zero. However, right after that we check that the `wethToDeposit` is not less than `MINIMUM_WETH_LIQUIDITY` which is more than zero in any case, so that check is redundant:

```
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          if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
```

```
12              revert TSwapPool__WethDepositAmountTooLow(
                   MINIMUM_WETH_LIQUIDITY, wethToDeposit);
13          }
14          ...
15      }
```

We can remove the `revertIfZero(wethToDeposit)` modifier from the function and save some gas costs.

[G-3] Unused local variable in the `TSwapPool::deposit` function could be removed

The variable `poolTokenReserves` in the `TSwapPool::deposit` function is not used in the function and could be removed to keep the code clean and save gas costs:

```
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              ...
14  ->          uint256 poolTokenReserves = i_poolToken.balanceOf(address(
        this));
15              ...
16          } else {
17              ...
18          }
19      }
```

[G-4] All `public` functions not used internally could be marked `external`

The functions `swapExactInput`and `swapExactOutput` in the `TSwapPool` contract are marked as **`public`** but are not used internally. It is recommended to mark them as `external` instead of **`public`** to save gas costs:

```
1       function swapExactInput(
2           IERC20 inputToken,
3           uint256 inputAmount,
4           IERC20 outputToken,
5           uint256 minOutputAmount,
```

```
6           uint64 deadline
7       )
8   ->      public
9           revertIfZero(inputAmount)
10          revertIfDeadlinePassed(deadline)
11          returns (uint256 output)
12      {
13          ...
14      }
```

The `swapExactOutput` function is used internally by the `sellPoolTokens`, but as it was suggested to delete it in the `[I-14]` issue, we can safely mark it as `external` as well.

[G-5] The `TSwapPool::totalLiquidityTokenSupply` function should either be marked as `external` and not used internally, or removed altogether

If it's used as `a more verbose way of getting the total supply of liquidity tokens` as stated in the natspec, it should be marked external and not used internally. Otherwise, it is better to just remove it, as it is quite self-explanatory that `totalSupply` returns the supply of the liquidity tokens and should be used. Removing this function would also save gas:

```
1       /// @notice a more verbose way of getting the total supply of
            liquidity tokens
2       function totalLiquidityTokenSupply() public view returns (uint256)
            {
3           return totalSupply();
4       }
```

Informational

[I-1] PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail. `PoolFactory` and `TSwapPool` contracts are using the "0.8.20" compiler version.

[I-2] Events are 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 of the fields should be indexed.

In the `PoolFactory` contract:

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

The `PoolCreated` event should have the `tokenAddress` and `poolAddress` fields indexed. In the `TSwapPool` contract:

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

The `Swap` event should have the `liquidityProvider`, `swapper`, `tokenIn`, and `tokenOut` fields indexed.

[I-3] The `PoolFactory` and `TSwapPool` contracts are missing natspec comments for the contracts and some of the functions

Natspec comments are missing for the `PoolFactory` and `TSwapPool` contracts and some of the functions. Natspec comments are used to provide a description of the contract and its functions. They are used to generate documentation for the contract and its functions. They are also used to provide information about the contract and its functions to developers who are reading the code.

Functions from the `PoolFactory` contract that are missing natspec comments:

```
    function createPool(address tokenAddress) external returns (address
        );
```

Functions from the `TSwapPool` contract that are missing natspec comments:

```
    function getOutputAmountBasedOnInput(
        uint256 inputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(inputAmount)
        revertIfZero(outputReserves)
        returns (uint256 outputAmount);
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
```

```
16          public
17          pure
18          revertIfZero(outputAmount)
19          revertIfZero(outputReserves)
20          returns (uint256 inputAmount);
21      function swapExactInput(
22          IERC20 inputToken,
23          uint256 inputAmount,
24          IERC20 outputToken,
25          uint256 minOutputAmount,
26          uint64 deadline
27      )
28          public
29          revertIfZero(inputAmount)
30          revertIfDeadlinePassed(deadline)
31          returns (uint256 output);
32      function swapExactOutput(
33          IERC20 inputToken,
34          IERC20 outputToken,
35          uint256 outputAmount,
36          uint64 deadline
37      )
38          public
39          revertIfZero(outputAmount)
40          revertIfDeadlinePassed(deadline)
41          returns (uint256 inputAmount);
42      function getPoolTokensToDepositBasedOnWeth(uint256 wethToDeposit)
           public view returns (uint256);
43      function _isUnknown(IERC20 token) private view returns (bool);
```

[I-4] The `PoolFactory` and `TSwapPool` contracts' elements does not follow the Solidity style guide order of layout

The elements of the `PoolFactory` and `TSwapPool` contracts should be laid out in the following order: - Type declarations - State variables - Events - Errors - Modifiers - Functions

Read more about the Solidity style guide order of layout here.

[I-5] Zero checks for addresses in the `PoolFactory` and `TSwapPool` contracts constructors are missing

It is recommended to check for zero addresses in the constructors of the contracts to prevent potential issues with zero addresses being passed to the contracts. In the `PoolFactory` contract:

```
1       constructor(address wethToken) {
2   ->      i_wethToken = wethToken;
```

```
3       }
```

And in the `TSwapPool` contract:

```
1       constructor(
2           address poolToken,
3           address wethToken,
4           string memory liquidityTokenName,
5           string memory liquidityTokenSymbol
6       )
7           ERC20(liquidityTokenName, liquidityTokenSymbol)
8       {
9   ->      i_wethToken = IERC20(wethToken);
10  ->      i_poolToken = IERC20(poolToken);
11      }
```

Also, it should be checked that the `poolToken` and `wethToken` addresses are not equal, as it makes no sense to create a pool with the same tokens.

[I-6] Wrong function call at `PoolFactory::createPool` resulting in a wrong LP token symbol

The `PoolFactory::createPool` function uses `.name()` function to get the name of the pool token for the LP token name. For some reason, it is also used to get the symbol of the LP token:

```
1       function createPool(address tokenAddress) external returns (address
            ) {
2           ...
3           string memory liquidityTokenName = string.concat("T-Swap ",
                IERC20(tokenAddress).name());
4   ->      string memory liquidityTokenSymbol = string.concat("ts", IERC20
        (tokenAddress).name());
5           ...
6       }
```

In the `PoolFactory::createPool` function, the LP token symbol suffix should be obtained using the `IERC20(tokenAddress).symbol()` function instead of `IERC20(tokenAddress).name()`.

[I-7] The `TSwapPool::TSwapPool__WethDepositAmountTooLow` error's variable `minimumWethDeposit` is actually a constant, and should not be emmitted

In the `TSwapPool::TSwapPool__WethDepositAmountTooLow` error, the constant minimumWethDeposit should not be emitted with the error message. The error should be defined as follows:

```
1     error TSwapPool__WethDepositAmountTooLow(uint256 wethToDeposit);
```

[I-8] Large literal values multiples of 10 can be replaced with scientific notation

Use `e` notation, for example: `1e18`, instead of its full numeric value. On lines 37 and 318 those should be replaced by:

```
1     ...
2     uint256 private constant MINIMUM_WETH_LIQUIDITY = 1e9;
3     ...
4             outputToken.safeTransfer(msg.sender, 1e18);
5     ...
```

[I-9] Misleading or confusing error messages

There are some error messages that could be misleading or confusing to the user. For example, the `TSwapPool::TSwapPool__MaxPoolTokenDepositTooHigh` error message could be misleading as it implies that the maximum pool token deposit is too high, when in fact it should indicate that we exceeded the maximum pool token deposit. The error message should be updated to:

```
1     error TSwapPool__MaxPoolTokenToDepositExceeded(uint256
          maximumPoolTokensToDeposit, uint256 poolTokensToDeposit);
```

Also the `TSwapPool::TSwapPool__MinLiquidityTokensToMintTooLow` error could be better renamed to:

```
1     error TSwapPool__MinLiquidityTokensToMintNotMet(uint256
          minimumLiquidityTokensToMint, uint256 liquidityTokensToMint);
```

And finally the `TSwapPool::TSwapPool__OutputTooLow` error could be better renamed to:

```
1     error OutputLowerThanExpectedMin(uint256 outputAmount, uint256
          minOutputAmount);
```

[I-10] Calculation logic could be moved to another function

As we already have a function `TSwapPool::getPoolTokensToDepositBasedOnWeth` to calculate our `poolTokensToDeposit` variable at `TSwapPool::deposit`, we could also seperate the logic for calculating the `liquidityTokensToMint` variable into a seperate `TSwapPool::getLiquidityTokensToMint` function. This would make the code more readable and easier to maintain.

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        ...
        if (totalLiquidityTokenSupply() > 0) {
            ...
            uint256 poolTokensToDeposit =
                getPoolTokensToDepositBasedOnWeth(wethToDeposit);
            ...
->          liquidityTokensToMint = wethToDeposit *
    totalLiquidityTokenSupply() / wethReserves;
            ...
        } else {
            ...
        }
    }
```

The function `TSwapPool::getLiquidityTokensToMint` would be defined as follows:

```
    function getLiquidityTokensToMint(uint256 wethToDeposit) internal
        pure returns (uint256) {
        uint256 wethReserves = i_wethToken.balanceOf(address(this));
        return wethToDeposit * totalLiquidityTokenSupply() /
            wethReserves;
    }
```

[I-11] The `TSwapPool::deposit` function does not pollow the `CEI` pattern

The `TSwapPool::deposit` function does not follow the `CEI` pattern:

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        ...
```

```
12          if (totalLiquidityTokenSupply() > 0) {
13              ...
14          } else {
15              _addLiquidityMintAndTransfer(wethToDeposit,
                    maximumPoolTokensToDeposit, wethToDeposit);
16  ->          liquidityTokensToMint = wethToDeposit;
17          }
18      }
```

The highlighted line should be moved before the `_addLiquidityMintAndTransfer` function call. It is not a state variable, but following `CEI` is still a best practice. You can read more about the `CEI` pattern here.

[I-12] Misleading natspec comment in `@dev` section of the `TSwapPool::_addLiquidityMintAndTransfer` function

There's a misleading natspec comment in the `@dev` section of the `TSwapPool::_addLiquidityMintAndTransfer` function:

```
1       /// @dev This is a sensitive function, and should only be called by
            addLiquidity
```

The actual function that calls the `_addLiquidityMintAndTransfer` function is the `deposit` function, not the `addLiquidity` function. The natspec comment should be updated to:

```
1       /// @dev This is a sensitive function, and should only be called by
            the deposit function
```

Or the `deposit` function should be renamed to `addLiquidity`, which would be more advisable.

[I-13] Usage of `magic numbers` is not advisable

The usage of `magic numbers` is not recommended as it makes the code less readable and harder to maintain. It is recommended to define constants for these values. For example, the `TSwapPool::getOutputAmountBasedOnInput` function uses the values `997` and `1000` as magic numbers:

```
1       ...
2       uint256 inputAmountMinusFee = inputAmount * 997;
3       uint256 numerator = inputAmountMinusFee * outputReserves;
4       uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
5       ...
```

It is recommended to define constants for these values:

```
    uint256 private constant FEE_MULTIPLIER = 997;
    uint256 private constant FEE_DIVIDER = 1000;
```

The same constants can be reused in `TSwapPool::getInputAmountBasedOnOutput` function. The `TSwapPool::_swap` function uses the value `1_000_000_000_000_000_000` as a magic number:

```
    ...
    outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
    ...
```

It is recommended to define a constant for this value:

```
    uint256 private constant EXTRA_REWARD = 1e18;
```

Also, the `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` use `1e18` as a magic number. It could be changed to:

```
    uint256 private constant ONE_UNIT = 1e18;
```

[I-14] The `TSwapPool::sellPoolTokens` function should be removed

The function `TSwapPool::sellPoolTokens` introduces a set of problems: - It sets the deadline instead of allowing the caller to set it - It is just a wrapper function over the `swapExactOutput` function, and we can't even set the deadline and the minimum expected output WETH amount, which are required for any swap call

This function should better be removed altogether.