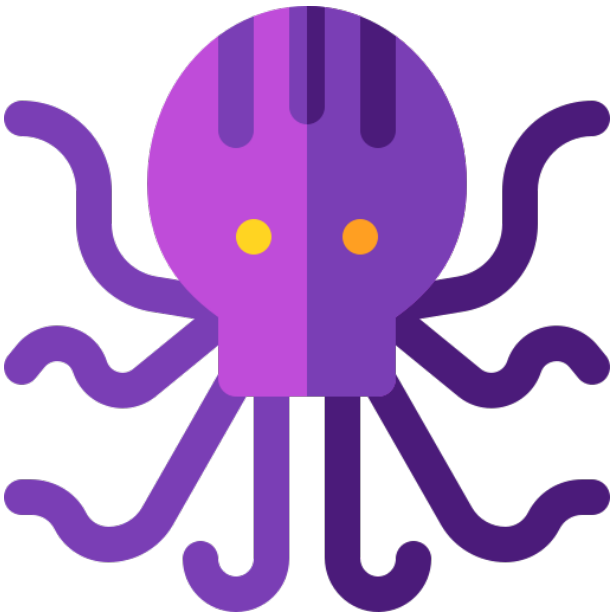

TSwap Initial Audit Report

Version 0.1

Yannick

May 22, 2024

TSwap Audit Report

Yannick

May 22, 2024

Prepared by: Yannick

Table of Contents

  - [L-1] `TSwapPool::_addLiquidityMintAndTransfer` emits the `LiquidityAdded` event with the arguments in a wrong order causing other services that read this event to misinterpret it
  - [L-2] The `TSwapPool::swapExactInput` default return value results in incorrect return value given

- Info
  - [I-1] SPDX-License misspelling
  - [I-2] Missing Zero Address Check in Constructor can lead to unexpected behavior
  - [I-3] Wrong implementation of liquidity token symbol
  - [I-4] Missing Zero Address Check in Constructor can lead to unexpected behavior
  - [I-5] Unused variable poolTokenReserves inside the deposit function bloats up the code
  - [I-6] Magic Numbers should be avoided for better code readability

- Gas Optimizations
  - [G-1] Unused Error Message Declaration bloat up code
  - [G-2] `TSwap::TSwapPool__WethDepositAmountTooLow` uses an unnecessary argument MINIMUM_WETH_LIQUIDITY

Disclaimer

The Security Research team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the 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

Audit Details

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

```
1ec3c30253423eb4199827f59cf564cc575b46db
```

Scope

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

Protocol Summary

TSwap 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.

The protocol starts as simply a `PoolFactory` contract. This contract is used to create new "pools" of tokens. It helps make sure every pool token uses the correct logic. But all the magic is in each `TSwapPool` contract.

You can think of each `TSwapPool` contract as it's own exchange between exactly 2 assets. Any ERC20 and the WETH token. These pools allow users to permissionlessly swap between an ERC20 that has a pool and WETH. Once enough pools are created, users can easily "hop" between supported ERC20s.

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.

Issues found

Severity	Number of issues found
High	5
Medium	0
Low	2
Info	6
Gas Optimizations	2
Total	15

Findings

High

[H-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 the deadline parameter is never used causing a transaction to be executed in the future, as a consequence a user could get bad market conditions if he wants to deposit liquidity.

Impact: Transactions could be send when the market conditions are really bad even when a deadline is specified.

Recommended Mitigation: Use the existing `TSwapPool::revertIfDeadlinePassed` modifier to check if the deadline is passed or not.

```
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      {}
```

[H-2] Incorrect fee calculation in `TSwap::getInputAmountBasedOnOutput` causes the protocol to charge way to much fees

Description: According to the Documentation "Every swap has a `0.3` fee, represented in `getInputAmountBasedOnOutput` and `getOutputAmountBasedOnInput`. Each applies a `997` out of `1000` multiplier." but the getInputAmountBasedOnOutput uses `10000` instead of `1000` for it's calculation.

Impact: The user must pay `91.3%` in fees instead of only `0.3`.

Proof of Concept: Copy this code in the `TSwapPool.t.sol` test file:

```
1       function testToMuchFeesCharged() public {
2           vm.startPrank(liquidityProvider);
3           weth.approve(address(pool), 100e18);
4           poolToken.approve(address(pool), 100e18);
5           pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
6           vm.stopPrank();
7
8           uint256 outputAmount = 1e17;
9           IERC20 inputToken = poolToken;
10          IERC20 outputToken = weth;
11          uint256 inputReserves = inputToken.balanceOf(address(pool));
12          uint256 outputReserves = outputToken.balanceOf(address(pool));
13          uint256 expectedSwapInputAmount =
14              ((inputReserves * outputAmount) * 1000) / ((outputReserves
                - outputAmount) * 997);
15
16          vm.startPrank(user);
17          poolToken.approve(address(pool), type(uint256).max);
18          poolToken.mint(user, 100e18);
19          uint256 inputAmountActual = pool.swapExactOutput(inputToken,
                outputToken, outputAmount, uint64(block.timestamp));
20          vm.stopPrank();
21
22          assertEq(inputAmountActual, expectedSwapInputAmount);
23      }
```

Recommended Mitigation: Replace the magic numbers with constant variables (as described in Issue [I-7]).

```
+ uint256 constant FEE_MULTIPLIER = 997;
+ uint256 constant BASE_MULTIPLIER = 1000;

- return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
    outputAmount) * 997);
+ return ((inputReserves * outputAmount) * BASE_MULTIPLIER) / ((
    outputReserves - outputAmount) * FEE_MULTIPLIER);
```

[H-3] Missing slipping protection in `TSwapPool::swapExactOutput` causes the user to pay way more `inputToken` as he might expect

Description: The `swapExactOutput` function has no `maxInputAmount` argument which can cause the user to pay way more tokens to receive his expected output if someone makes a huge swap before him. If his transaction is pending in the mempool someone could make a swap before him manipulating the swap rate for the user.

Impact: If the market conditions change the user could get a much worse swap than he might expect.

Proof of Concept: First of all change the `getInputAmountBasedOnOutput` to reflect the correct base multiplier as described in [H-2] `Incorrect fee calculation`.

```
+ uint256 constant FEE_MULTIPLIER = 997;
+ uint256 constant BASE_MULTIPLIER = 1000;

- return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
    outputAmount) * 997);
+ return ((inputReserves * outputAmount) * BASE_MULTIPLIER) / ((
    outputReserves - outputAmount) * FEE_MULTIPLIER);
```

Then copy the following code inside the `TSwapPool.t.sol` file:

```
    function testMissingSlippageProtection() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 priceOfOneWethInPoolTokensBefore = pool.
            getPriceOfOneWethInPoolTokens();
        // The price of one weth in pool tokens is =
            0.987158034397061298

        uint256 outputAmountAttacker = 50e18;
```

```
12          uint256 outputAmountUser = 10e18;
13          IERC20 inputToken = weth;
14          IERC20 outputToken = poolToken;
15          uint256 inputReserves = inputToken.balanceOf(address(pool));
16          uint256 outputReserves = outputToken.balanceOf(address(pool));
17          uint256 expectedInputAmount =
18              ((inputReserves * outputAmountUser) * 1000) / ((
                    outputReserves - outputAmountUser) * 997);
19          // expectedInputAmount = 11.144544745347152568 WETH for 10 pool
                tokens
20
21          vm.startPrank(attacker);
22          weth.approve(address(pool), 1000e18);
23          pool.swapExactOutput(inputToken, outputToken,
                outputAmountAttacker, uint64(block.timestamp));
24          vm.stopPrank();
25
26          uint256 priceOfOneWethInPoolTokensAfter = pool.
                getPriceOfOneWethInPoolTokens();
27          // now the user only gets 0.247642917930848648 pool tokens for
                1 WETH
28
29          vm.startPrank(user);
30          weth.mint(user, 1000e18);
31          weth.approve(address(pool), 1000e18);
32          uint256 actualInputAmount = pool.swapExactOutput(inputToken,
                outputToken, 10e18, uint64(block.timestamp));
33          // actualInputAmount = 50.225903387192671293 WETH to get 10
                pool tokens
34          vm.stopPrank();
35
36          //? so the user has to pay ~50 WETH instead of ~11 WETH to get
                his 10 pool tokens
37
38          console2.log("priceOfOneWethInPoolTokensBefore",
                priceOfOneWethInPoolTokensBefore);
39          console2.log("priceOfOneWethInPoolTokensAfter",
                priceOfOneWethInPoolTokensAfter);
40
41          assertEq(actualInputAmount, expectedInputAmount);
42      }
```

Recommended Mitigation: Add a `maxInputAmount` argument to the `swapExactOutput` function, similar to the `swapExactInput` argument `minOutputAmount`. With this argument a user can specify what he is willing to pay at a maximum if the market conditions change. Then add a condition that the transaction should revert if the calculated `inputAmount` is higher as the `maxInputAmount` .

```
1       function swapExactOutput(
2           IERC20 inputToken,
```

```
3           IERC20 outputToken,
4           uint256 outputAmount,
5           uint64 deadline,
6   +       uint256 maxInputAmount
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
18  +       if(inputAmount > maxInputAmount){
19  +          revert();
20  +       }
21
22          _swap(inputToken, inputAmount, outputToken, outputAmount);
23      }
```

[H-4] `TSwapPool::sellPoolTokens` uses `swapExactOutput` instead of `swapExactInput` causing the user to receive the wrong amount of tokens

Description: The usage of the `swapExactOutput` function is wrong here, according to the documentation of the `swapExactOutput` function it "figures out how much you need to input based on how much output you want to receive.". The `sellPoolTokens` function on the other side is used to sell a specific amount of pool tokens based on the user input and should therefore use `swapExactInput`.

Impact: The user will swap the wrong amount of tokens, which is a disruption in the protocols functionality.

Proof of Concept: Copy the following code into `TSwapPool.t.sol`:

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

```
11          uint256 expectedWethAmount =
12              pool.getOutputAmountBasedOnInput(poolTokenAmountToSell,
                    inputReserves, outputReserves);
13
14          vm.startPrank(user);
15          weth.mint(user, 100e18);
16          poolToken.mint(user, 100e18);
17          poolToken.approve(address(pool), 100e18);
18          weth.approve(address(pool), 100e18);
19          uint256 actualWethAmountReceived = pool.sellPoolTokens(
                poolTokenAmountToSell);
20          vm.stopPrank();
21
22          assertEq(actualWethAmountReceived, expectedWethAmount);
23      }
```

Recommended Mitigation: This change also assumes that the `swapExactInput` function returns the unused return value `uint256 output`. Change `swapExactOutput` to `swapExactInput`, additionally to that change the `sellPoolTokens` functions need to accept two new parameter `uint256 minOutputAmount, uint64 deadline` to pass these to `swapExactInput` in order to work correct.

```
1       function sellPoolTokens(
2           uint256 poolTokenAmount,
3   +       uint256 minOutputAmount,
4   +       uint64 deadline
5           ) external returns (uint256 wethAmount) {
6   -           return swapExactOutput(i_poolToken, i_wethToken,
        poolTokenAmount, uint64(block.timestamp));
7   +           return swapExactInput(i_poolToken, poolTokenAmount, i_wethToken
        , minOutputAmount, deadline);
8       }
```

[H-5] `TSwapPool::_swap` has unnecessary logic to give every 10 swaps the user extra tokens breaking the protocols invariant

Description: According to the Documentation the protocol has a strict invariant `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 ratio is broken because of the following code sending every 10 swaps additional tokens to the user:

```
1       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 abuse this to send multiple transactions draining the pool over time and collecting the extra incentive.

Proof of Concept:

A user swaps 10 times to collect the 1_000_000_000_000_000_000 extra tokens and keep going until all funds are drained.

```
    function testInvariantBroken() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 outputWeth = 1e17;

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

        uint8 numberOfSwaps = 9;

        for (uint8 i = 0; i < numberOfSwaps; i++) {
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
        }

        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);

        // The 10th swap breaks the invariant
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp));
        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);
        assertEq(actualDeltaY, expectedDeltaY);
    }
```

Recommended Mitigation: Remove the incentive mechanism or set aside tokens the same way we do with fees.

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

Low

[L-1] `TSwapPool::_addLiquidityMintAndTransfer` emits the `LiquidityAdded` event with the arguments in a wrong order causing other services that read this event to misinterpret it

Description: The event should be emitted with these arguments in this order `event LiquidityAdded(address indexed liquidityProvider, uint256 wethDeposited, uint256 poolTokensDeposited);` but the `_addLiquidityMintAndTransfer` provides the poolTokensToDeposit as second argument and then the wethToDeposit which is wrong.

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

Recommended Mitigation:

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

[L-2] The `TSwapPool::swapExactInput` default return value results in incorrect return value given

Description: The `swapExactInput` should return the correct amount of output tokens a user received. However the named return value `ouput` is never assigned a value.

Impact: The return value will always be 0 giving a wrong information to the caller of the function.

Proof of Concept: Manual review

Recommended Mitigation:

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
```

```
6           uint64 deadline
7       )
8           public
9           revertIfZero(inputAmount)
10          revertIfDeadlinePassed(deadline)
11          returns (
12              uint256 output
13          )
14      {
15          uint256 inputReserves = inputToken.balanceOf(address(this));
16          uint256 outputReserves = outputToken.balanceOf(address(this));
17
18  -       uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
        inputReserves, outputReserves);
19  +       output = getOutputAmountBasedOnInput(inputAmount, inputReserves
        , outputReserves);
20
21  -       if (outputAmount < minOutputAmount) {
22  -           revert TSwapPool__OutputTooLow(outputAmount,
        minOutputAmount);
23  +       if (output < minOutputAmount) {
24  +           revert TSwapPool__OutputTooLow(output, minOutputAmount);
25          }
26
27  -       _swap(inputToken, inputAmount, outputToken, outputAmount);
28  +       _swap(inputToken, inputAmount, outputToken, output);
29      }
```

Info

[I-1] SPDX-License misspelling

Description: The SPDX-License-Identifier inside `PoolFactory.sol` is wrong spelled.

Impact:

If the SPDX-License-Identifier is missing in a Solidity file, it can lead to several issues. Firstly, the licensing terms become unclear, creating legal uncertainty for users and developers. This ambiguity can cause compliance problems with open-source policies, preventing organizations from using or contributing to the code. Additionally, automated tools for license detection and compliance checking may not function correctly, disrupting development workflows. The absence of a clear license can also deter developers from engaging with the project, hindering its growth and collaboration within the community. Including the SPDX-License-Identifier ensures clarity, legal compliance, and smoother integration with development tools.

Recommended Mitigation:

```
- // SPDX-License-I   ntifier: GNU General Public License v3.0
+ // SPDX-License-Identifier: GNU General Public License v3.0
```

[I-2] Missing Zero Address Check in Constructor can lead to unexpected behavior

Description: The constructor of the `PoolFactory` contract assigns the `wethToken` parameter directly to the `i_wethToken` variable without performing a check to ensure that the provided address is not the zero address.

Impact: Failing to check for the zero address (`address(0)`) could potentially allow the deployment of the contract with an invalid or uninitialized `i_wethToken` address, leading to unexpected behavior or vulnerabilities during contract execution.

Recommended Mitigation: Implement a require statement in the constructor to check that the provided wethToken address is not the zero address. This will ensure that the contract is initialized with a valid address, reducing the risk of unexpected behavior or vulnerabilities.

```
    constructor(address wethToken) {
+       require(wethToken != address(0), "Zero address detected");
        i_wethToken = wethToken;
     }
```

[I-3] Wrong implementation of liquidity token symbol

Description: In the `PoolFactory::createPool` function the liquidityTokenSymbol gets assigned by calling `IERC20(tokenAddress).name()` but this should be the symbol not the name.

Impact: Token symbol is way to long.

Recommended Mitigation:

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

[I-4] Missing Zero Address Check in Constructor can lead to unexpected behavior

Description: The constructor of the `TSwapPool` contract assigns the `wethToken` parameter directly to the `i_wethToken` variable without performing a check to ensure that the provided address is not the zero address. The same happens for the `poolToken`.

Impact: Failing to check for the zero address `address(0)` could potentially allow the deployment of the contract with an invalid or uninitialized `i_wethToken` and `i_poolToken` address, leading to unexpected behavior or vulnerabilities during contract execution.

Recommended Mitigation: Implement a require statement in the constructor to check that the provided wethToken and i_poolToken address is not the zero address. This will ensure that the contract is initialized with a valid address, reducing the risk of unexpected behavior or vulnerabilities.

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

[I-5] Unused variable poolTokenReserves inside the deposit function bloats up the code

Description: The poolTokenReserves variable is not used and should be removed.

Impact: It costs more gas to deploy that contract and makes the code harder to understand / read.

Recommended Mitigation:

```
- uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

[I-6] Magic Numbers should be avoided for better code readability

Description: Magic numbers should be defined as constants with a declarative name describing the value.

Impact: Developers can get confused about the meaning of a magic number, making the code harder to read and maintain.

Recommended Mitigation:

Declare numbers as constants.

```
+ uint256 constant FEE_MULTIPLIER = 997;
+ uint256 constant BASE_MULTIPLIER = 1000;
+ uint256 constant ONE_WETH_IN_WEI = 1e18;

- uint256 inputAmountMinusFee = inputAmount * 997;
+ uint256 inputAmountMinusFee = inputAmount * FEE_MULTIPLIER;

- uint256 denominator = (inputReserves * 1_000) + inputAmountMinusFee;
+ uint256 denominator = (inputReserves * BASE_MULTIPLIER) +
    inputAmountMinusFee;

- return ((inputReserves * outputAmount) * 10_000) / ((outputReserves -
    outputAmount) * 997);
+ return ((inputReserves * outputAmount) * BASE_MULTIPLIER) / ((
    outputReserves - outputAmount) * FEE_MULTIPLIER);

-  return getOutputAmountBasedOnInput(
-  1e18,
-  i_wethToken.balanceOf(address(this)),
-  i_poolToken.balanceOf(address(this))
-  );
+  return getOutputAmountBasedOnInput(
+  ONE_WETH_IN_WEI,
+  i_poolToken.balanceOf(address(this)),
+  i_wethToken.balanceOf(address(this))
+  );

-  return getOutputAmountBasedOnInput(
-  1e18,
-  i_wethToken.balanceOf(address(this)),
-  i_poolToken.balanceOf(address(this))
-  );
+  return getOutputAmountBasedOnInput(
+  ONE_WETH_IN_WEI,
+  i_poolToken.balanceOf(address(this)),
+  i_wethToken.balanceOf(address(this))
+  );
```

Gas Optimizations

[G-1] Unused Error Message Declaration bloat up code

Description: The error message `PoolFactory::PoolFactory__PoolDoesNotExist` is declared but appears to be unused within the contract code.

Impact: While unused error messages don't directly affect the functionality or security of the contract, they can lead to confusion for developers reviewing the code. They may wonder about the

purpose of the declared error message, potentially leading to misunderstandings during maintenance or debugging.

Recommended Mitigation:

Remove the declaration of the unused error message to improve code clarity and avoid confusion for developers. If the error message is intended for future use, document its purpose and potential scenarios where it might be utilized.

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

[G-2] `TSwap::TSwapPool__WethDepositAmountTooLow` uses an unnecessary argument `MINIMUM_WETH_LIQUIDITY`

Description: The `TSwapPool::WethDepositAmountTooLow` error use the MINIMUM_WETH_LIQUIDITY constant as argument but this can be read through the `getMinimumWethDepositAmount` function, so it's not needed and the protocol could safe gas here.

Impact: It costs more gas to emit that error.

Recommended Mitigation:

Remove the argument from the event and update all references.

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