TSwap Protocol Audit Report

Version 1.0

github.com/lucasfhope

October 3, 2025

TSwap Protocol Audit Report

Lucas Hope

October 3, 2025

Prepared by: Lucas Hope

Table of Contents

Protocol Summary

The protocol intends to be a decentralized exchange which uses a constant product formula to set the swap rate between WETH and a pool token. The pool token can be set as any ERC20 during pool creation. The DEX is provided funds by liquidity providers that receive liquidity tokens to represent their stake in the pool. Users can swap between tokens in a pool but each swap will include a 0.3% fee.

Disclaimer

Lucas Hope 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 report corresponds to the following commit hash:

```
f5a14bc4c1f202aa2f8bdc210cdfea7e1f276735
```

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

The audit found issues with certain function in the protocol that were not implemented as the protocol intended. The audit also found code and other instances that cause the core invaraint of the protocol to break.

Issues found

Severity	Number of issues found
High	4
Medium	2
Low	2
Info	7
Gas	3
Total	**18**

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes the protocol to take took many tokens from the users, resulting in fees that are too high.

Description: The `TSwapPool::getInputAmountBasedOnOutput` function is intended to calculate the the amount of tokens should be deposited for a given output of tokens, which should include a 0.3% fee. Instead, the function calculates the the fee at over 90%. This is because it scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Proof of Code: You can add the following test to your test suite in `test/unit/TSwapPool.t.test`.

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

    // expected swap amount with 0.3% fee, want to swap ?? PoolToken
       for 1 WETH
    uint256 expectedPoolTokens = ((poolToken.balanceOf(address(pool)) *
        1 ether) * 1000) / ((weth.balanceOf(address(pool)) - 1 ether) *
        997);  // correct fee ratio -> 997/1000
    uint256 calculatedPoolTokens = pool.getInputAmountBasedOnOutput(
        1 ether,
        poolToken.balanceOf(address(pool)),
        weth.balanceOf(address(pool))
    );

    console.log("expectedPoolTokens:   ", expectedPoolTokens);
    console.log("calculatedPoolTokens: ", calculatedPoolTokens);
    assert(expectedPoolTokens != calculatedPoolTokens);
}
```

When you run the test, you will see that the calculated amount of tokens is 10 times larger than what it should be.

Recommended Mitigation:

```
function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(outputAmount)
    revertIfZero(outputReserves)
    returns (uint256 inputAmount)
{
-        return ((inputReserves * outputAmount) * 10000) / ((
    outputReserves - outputAmount) * 997);
+        return ((inputReserves * outputAmount) * 1000) / ((
    outputReserves - outputAmount) * 997);
}
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` could cause users to receive way fewer tokens than expected.

Description: The `TSwapPool::swapExactOutput` function does not inlcude any sort of slippage protection. This function is similar to swap is done in `TSwapPool::swapExactInput`, where the function specifies `minOutputAmount`. The `TSwapPool:swapExactOutput` function should specify a `maxInputAmount`.

Impact: If the market conditions chnage before the transaction process, the user would get a much worse swap.

Proof of Concept: 1. The price of 1 WETH right now is 1000 USDC 2. User inputs a `swapExactOutput` looking for 1 WETH 1. inputToken = USDC 2. outputToken = WETH 3. outputAmount = 1 4. dealine = anything 3. The function does not offer a max input amount 4. As the transaction is pending in the mempool, the market changes! The price moves a lot such that 1 WETH -> 3000 USDC, 3x more than the user expected. 5. The transaction completes but the user sends the protocol 10,000 USDC instead of 1000 USDC.

Proof of Code: You can add the following test to your test suite in `test/unit/TSwapPool.t.test`. You should also update `TSwapPool::getInputAmountBasedOnOutput` so that the fee ratio is 997 / 1000 instead of 997 / 10000.

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

        uint256 outputAmount = 5 ether;
        uint256 inputReserves = weth.balanceOf(address(pool));
        uint256 outputReserves = poolToken.balanceOf(address(pool));

        // we want to get 5 WETH, how many PoolTokens do we need to
            give
        uint256 expectedInputAmount = pool.getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

        // another user does a swap before to change the pool state
        address user2 = makeAddr("user2");
        weth.mint(user2, 50e18);
        vm.startPrank(user2);
        weth.approve(address(pool), 50e18);
        pool.swapExactInput(
            weth,
            50 ether,
            poolToken,
            0,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        // user expects to receive 5 WETH for expected amount of
            PoolTokens
        weth.mint(user, 100e18); // mint more WETH to user to ensure
            they have enough
        uint256 userStartingPoolTokenBalance = poolToken.balanceOf(user
            );
        uint256 userStartingWethBalance = weth.balanceOf(user);
        vm.startPrank(user);
        weth.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(
            weth,
            poolToken,
            outputAmount,
            uint64(block.timestamp) + 1
        );
        uint256 userEndingPoolTokenBalance = poolToken.balanceOf(user);
        uint256 userEndingWethBalance = weth.balanceOf(user);
        vm.stopPrank();

        uint256 userPoolTokensReceived = userEndingPoolTokenBalance -
            userStartingPoolTokenBalance;
        uint256 userWethSpent = userStartingWethBalance -
            userEndingWethBalance;

        console.log("Pool Tokens Received:  ", userPoolTokensReceived);
        console.log("Pool Tokens Expected:  ", outputAmount);
        console.log("WETH Expected to Send: ", expectedInputAmount);
        console.log("WETH Spent:            ", userWethSpent);

        assert(userWethSpent > expectedInputAmount);

    }
```

You will see that the user now has to spend more that double the amount of WETH to get the same amount of pool tokens.

Recommended Mitigation: Update `swapExactOutput` to include slippage protection.

```
function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxInputAmount
        uint64 deadline
    )
        public
        revertIfZero(outputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 inputAmount)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

        inputAmount = getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

+       if(inputAmount > maxInputAmount) {
+           revert();
+       }

        _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 is intended to allow users to easily sell pool tokens and receive WETH in exchnage. Users indicate how many pool tokens they are willing to sell in the 'poolTokenAmount parameter, but the function currently miscalculates the swapped amount.

This is because the `swapExactOutput` function is called whereas the `swapExactInput` function is the one that should be called since the users are specifying the exact amount of input pool tokens.

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

Proof of Code: You can add the following test to your test suite in `test/unit/TSwapPool.t.test`.

```
function testSellPoolTokensMishandlesSwapCalculation() public {
    // LP deposit liquidity, 2 to 1 ratio weth to PT
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 50e18, uint64(block.timestamp));
    vm.stopPrank();

    // user attempts to swap 10 PT for WETH, should get around 20 WETH
    uint256 startingBalanceWeth = weth.balanceOf(user);
    uint256 startingBalancePoolToken = poolToken.balanceOf(user);

    vm.startPrank(user);
    uint256 poolTokensToSell = 10 ether;
    poolToken.approve(address(pool), type(uint256).max);
    pool.sellPoolTokens(poolTokensToSell);
    vm.stopPrank();

    uint256 endingBalanceWeth = weth.balanceOf(user);
    uint256 endingBalancePoolToken = poolToken.balanceOf(user);
    uint256 wethReceived = endingBalanceWeth - startingBalanceWeth;
    uint256 poolTokensSpent = startingBalancePoolToken -
        endingBalancePoolToken;

    console.log("WETH Received:       ", wethReceived);
    console.log("Pool Tokens Spent:   ", poolTokensSpent);
    console.log("Pool Tokens To Sell: ", poolTokensToSell);


    assert(poolTokensSpent != poolTokensToSell);
}
```

You will see that while the user intended to sell 10 pool tokens, he instead only sold around 5.5 pool tokens while receiving 10 weth instead.

Recommended Mitigation: Update `sellPoolTokens` suhc that it uses `swapExactInput` rather than `swapExactOutput`.

```
    function sellPoolTokens(
```

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

[H-4] In `TSwapPool::_swap`, the extra sent to users after every 10 swaps 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 ration between the two amounts should remain constant, hence the `k`. However, this is broken when teh `_swap` function sends extra tokens every 10 swaps, meaning that over time, the protocol funds will be drained.

Impact: A user could maliciously drain the funds of the protocol by doing a lot of swaps and collecting the extra incentive given out by the protocol. The protocol's core invariant is broken.

Proof of Code: You can add the following test to your test suite in `test/unit/TSwapPool.t.test`.

```
function testSwapBreaksInvariant() 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);
        for(uint256 i = 0; i < 9; i++) {
            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();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);
        console.log("ExpectedDeltaY: ", expectedDeltaY);
        console.log("ActualDeltaY:   ", actualDeltaY);
        assert(expectedDeltaY != actualDeltaY);
}
```

This test will show, after the tenth swap, extra tokens will be sent to the user and the change in WETH tokens exceeds the expected change.

Recommended Mitigation: You should remove the logic in `_swap` that sends extra tokens every 10 swaps.

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

    ...

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

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

```
        emit Swap(
            msg.sender,
            inputToken,
            inputAmount,
            outputToken,
            outputAmount
        );

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

Medium

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

Description: The `deposit` function accepts a dealine parameter, which is the deadline for the transaction to be completed by in the documentation. This parameter is never used.

Impact: Transactions can be completed after the dealine that the function caller had set. Therefore, operations that add liquidity to the pool might be executed at unexpected times and in market conditions where the deposit rate is unfavorable. This also makes the transaction susceptible to MEV attacks.

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

Recommended Mitigation: Make the following chnage to the function within the function header.

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

[M-2] Rebase, fee-on-transfer, and ERC777 tokens break protocol invariant.

Description: Rebase, fee-on-transfer, and ERC777 are weird ERC20s that will break the protocol invariant if they are deployed as the pool token in a TSwapPool.

Rebase tokens will update their value over time without transfers. A positive rebase will create a surplus of tokens and a negative rebase can make reserves lower than assumed, causing swaps to revert or set prices incorrectly.

Fee-on-transfer tokens will take some of the token balance when the token is transfered. The protocol will calculate the swap without accouning for the fee and break the protocol invariant.

ERC777 tokens leave the protocol open to reentrancies and should be avoided.

Impact: All of these weird ERC20 tokens will make your protocol vulnerable and break the core invariant of the protocol.

Proof of Code: This example will show how a fee-on-transfer token will break the protocol invariant. First, you will need a FeeOnTranferToken contract.

```
contract FeeOnTransferToken is ERC20 {
    constructor() ERC20("Rebase", "RT") {}

    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }

    // Will take a 1e18 fee on every transfer
    function _update(address from, address to, uint256 value) internal
        virtual override {
        // Skip fee on mint/burn (from==0 or to==0)
        if (from != address(0) && to != address(0) && value > 0) {
            super._update(from, address(0), 1e18);
            super._update(from, to, value);
        } else {
            super._update(from, to, value);
        }
    }
}
```

Now, You can add the following test to your test suite in `test/unit/TSwapPool.t.test`.

```
function testFeeOnTransferTokenBreakInvariant() public {
    FeeOnTransferToken feeToken = new FeeOnTransferToken();
    pool = new TSwapPool(address(feeToken), address(weth), "Fee", "FT")
        ;

    feeToken.mint(liquidityProvider, 200e18);
    feeToken.mint(user, 100e18);
    weth.mint(user, 100e18);

    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    feeToken.approve(address(pool), 101e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    vm.startPrank(user);
    weth.approve(address(pool), type(uint256).max);
    uint256 outputFeeToken = 1e18;
    int256 startingX = int256(feeToken.balanceOf(address(pool)));
    int256 expectedDeltaX = int256(-1) * int256(outputFeeToken);
    pool.swapExactOutput( weth, feeToken, outputFeeToken, uint64(block.timestamp));
    vm.stopPrank();

    uint256 endingX = feeToken.balanceOf(address(pool));
    int256 actualDeltaX = int256(endingX) - int256(startingX);
    console.log("ExpectedDeltaX: ", expectedDeltaX);
    console.log("ActualDeltaX:   ", actualDeltaX);
    assert(expectedDeltaX != actualDeltaX);
}
```

Now you will see that the fee-on-transfer pool token will break the protocol invariant by transfering an extra token during the swap.

Recommended Mitigation: Do not create a pool with a rebase, fee-on-tranfer, or ERC777 token.

Low

[L-1] `TSwapPool::LiquidityAdded` event in `TSwapPool::_addLiquidityMintAndTransfer` has parameters out of order.

Description: When the `LiquidityAdded` event is emitted in `TSwapPool::_addLiquidityMintAndTransf`, it logs value in an incorrect order. The `poolTokensToDeposit` value should go in the third parameter position, while the `wethToDeposit` value should go seciond.

Impact: Off chain functions may potentially malfunction do the incorrect event emission.

Recommended Mitigation: Update the parameters of the `LiquidityAdded` event emission in `TSwapPool::_addLiquidityMintAndTransfer`.

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

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

Description: The `TSwapPool::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 and there is not return statement used.

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

Proof of Code: You can add the following test to your test suite in `test/unit/TSwapPool.t.test`.

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

    uint256 inputAmount = 1 ether;
    uint256 inputReserves = poolToken.balanceOf(address(pool));
    uint256 outputReserves = weth.balanceOf(address(pool));

    uint256 expectedOutputAmount = pool.getOutputAmountBasedOnInput(
        inputAmount,
        inputReserves,
        outputReserves
    );

    vm.startPrank(user);
    poolToken.approve(address(pool), inputAmount);
    uint256 swapReturnedOutputAmount = pool.swapExactInput(
        poolToken,
        inputAmount,
        weth,
        expectedOutputAmount,
        uint64(block.timestamp)
    );
    vm.stopPrank();

    console.log("expectedOutputAmount:     ", expectedOutputAmount);
    console.log("swapReturnedOutputAmount: ", swapReturnedOutputAmount)
       ;

    assert(swapReturnedOutputAmount == 0);
}
```

You will be able to see that the returned output from `swapExactInput` is 0.

Recommended Mitigation: You should either update the returned value `output` to `outputAmount` or change every instance of `outputAmount` in the function code to `output`.

```
    function swapExactInput(
        IERC20 inputToken,    // e input tokens to swap / sell
        uint256 inputAmount,  // e amount of input tokens to swap /
            sell
        IERC20 outputToken,   // e output tokens to receive / buy
        uint256 minOutputAmount, // e minimum amount of output tokens
            to receive / buy
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        // @audit-low return not used, incorrect name
-       returns (uint256 output)
+       return (uint256 outputAmount)
    {
        ...
    }
```

Informational

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` error is defined but never used and should be removed.

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

[I-2] Missing zero address checks in the constructors of `PoolFactory` and `TSwapPool` when setting the token addresses.

You should make sure you are checking that the token addresses are not being set to the zero address.

`PoolFactory`:

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

`TSwapPool`:

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

[I-3] `PoolFactory::liquidityTokenSymbol` should use `.symbol()` instead of `.name()` when creating the symbol for the liquidity provider token.

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

[I-4] `TSwapPool::deposit` updates internal state after an external call, which does not follow CEI.

Move the internal state change of `liquidityTokensToMint`, which is the return value, before the external calls in `_addLiquidityMintAndTransfer` to follow best practices.

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

[I-5] "Magic" numbers in `TSwapPool` should be set as constants.

`TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool:getInputAmountBasedOnOutput` both contain literals that should be instead defined as constants.

For example, in `TSwapPool::getOutputAmountBasedOnInput`, you have:

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

Instead, these values should be defined as constants at the beginning of the contract:

```
uint256 public constant FEE_ADJUSTMENT= 997;
uint256 public constant FEE_PRECISION = 1000;

...

uint256 inputAmountMinusFee = inputAmount * FEE_ADJUSTMENT;
uint256 numerator = inputAmountMinusFee * outputReserves;
uint256 denominator = (inputReserves * FEE_PRECISION) +
    inputAmountMinusFee;
return numerator / denominator;
```

This should be done for other instances of "magic" numbers in the contract.

[I-6] `TSwapPool::swapExactInput` does not have any natspec.

You should have natspec explaining what the `TSwapPool::swapExactInput` function does and what the input parameters are as one of the main contract functions used by external entities.

[I-7] `TSwapPool::swapExactOutput` natspec misses the `deadline` parameter.

You should add natspec for the `TSwapPool::swapExactOutput` function to explain the deadline parameter.

Gas

[G-1] `TSwapPool::MINIMUM_WETH_LIQUIDITY` is a constant and should not be emitted within a revert.

To save some gas on reverts, `TSwapPool::TSwapPool__WethDepositAmountTooLow` should only contain the paremeter for the WETH amount that was too low to deposit.

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

    ...

    revert TSwapPool__WethDepositAmountTooLow(
-       MINIMUM_WETH_LIQUIDITY,
        wethToDeposit
    );
```

[G-2] `TSwapPool::deposit` contains an unused variable, which should be removed to save gas.

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

[G-3] `TSwapPool::swapExactInput` should be marked as external.

`TSwapPool::swapExactInput` is never called internally, so you can save gas by marking as external.