T-Swap Audit Report

Lead Auditors:

- Benas Volkovas

Table of Contents

Disclaimer

I make 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 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

I 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 codebase: https://github.com/Cyfrin/5-t-swap-audit.

And commit hash: `f4337615b7eada3d871378ff8b70af08277d3dec`

Scope

src/

--- PoolFactory.sol

--- TSwapPool.sol

Protocol Summary

This project is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX). T-Swap is known as an Automated Market Maker (AMM) because it doesn't use a normal "order book" style exchange, instead it uses "Pools" of an asset.

Roles

- User: Anyone who interacts with the protocol.

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Info	15
Gas Optimizations	5
Total	27

Findings

High

[H-1] Incorrect fee calculation causes the protocol to take fee of `90.03%` instead of `0.3%` and take funds from users

Description: in `TSwapPool::getInputAmountBasedOnOutput` , the `10000` is used as a "magic number" to calculate the input amount. This "magic number" is incorrect and should be `1000` . Incorrect "magic number" causes the protocol to take fee and the user to get only `9.97%` of the output amount instead of `99.7%` .

Impact: Users lose `90.03%` of value when swapping tokens

Proof of Concept:

1. Liquidity provider deposits `100'000 USDC` and `100 WETH` to the pool
2. `1 WETH` is worth `1000 USDC`
3. User calls `TSwapPool::swapExactOutput` with `outputAmount` set to `1 WETH`
4. `getInputAmountBasedOnOutput` calculates the USDC user will need to sell in order to buy `1 WETH`
   1. `inputReserves` is `100'000 USDC`
   2. `outputReserves` is `100 WETH`
   3. `outputAmount` is `1 WETH`
   4. `((inputReserves * outputAmount) * 10000) / ((outputReserves - outputAmount) * 997)`
   5. `((100'000 * 1) * 10000) / ((100 - 1) * 997)`
   6. `10131.404314 ~= 10131 USDC`
5. User sells `10131 USDC` for `1 WETH`

POC code

Paste this test in `test/unit/TSwapPool.t.sol` directory

```
function test_getInputAmountBasedOnOutput_CalculatesTheInputIncorrectly()
    external
{
    uint256 initialBalance = 1_000_000 ether;
    uint256 wethReserves = 100 ether;
    uint256 poolTokenReserves = 100_000 ether;
    uint256 wethBuyAmount = 1 ether;
    uint256 expectedPoolTokensSellAmount = ((poolTokenReserves *
        wethReserves) * 1000) / ((wethReserves - wethBuyAmount) * 997);

    // Setup tokens and pool
    poolToken = new ERC20Mock();
    weth = new ERC20Mock();
    pool = new TSwapPool(
        address(poolToken),
        address(weth),
        "LTokenA",
        "LA"
    );

    // Mint tokens for liquidity provider and usr
    weth.mint(liquidityProvider, initialBalance);
    poolToken.mint(liquidityProvider, initialBalance);

    weth.mint(user, initialBalance);
    poolToken.mint(user, initialBalance);

    // Add liquidity (100_000 pool tokens, 100 weth)
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), wethReserves);
    poolToken.approve(address(pool), poolTokenReserves);
    pool.deposit(
        wethReserves,
        wethReserves,
        poolTokenReserves,
        uint64(block.timestamp)
    );
    vm.stopPrank();

    // Try to get input amount based on output
    vm.startPrank(user);
```

```
        uint256 poolTokensSellAmount = pool.getInputAmountBasedOnOutput(
            wethBuyAmount,
            poolTokenReserves,
            wethReserves
        );

        assertEq(
            poolTokensSellAmount,
            expectedPoolTokensSellAmount,
            "Incorrect pool tokens sell amount"
        );
    }
```

Recommended Mitigation: Change the "magic number" `10000` to `1000`. Define constant variable at the beginning of the contract.

[H-2] Lack of slippage protection in `TSwapPool::swapExactInput` allows to front-run transactions and increase the price of the token.

Description: In `swapExactInput`, the `maxInputAmount` parameter is not present and users cannot protect themself from slippage. This means that the user is not protected from price changes. This allows the attacker to front-run the transaction and increase the price of the token, which will cause the user to spend more than they wanted. After the victim's transaction is mined, the attacker can sell all the tokens they bought and make profit.

Impact: If market conditions change before the transaction is processed, the user might get a much worse swap and lose money.

Proof of Concept:

1. User calls `TSwapPool::swapExactOutput` with `outputAmount` set to `1000 USDC`
2. Attacker sees the transaction in the mempool and buys 90% of the `USDC` tokens from the pool
3. The price of `USDC` token increases
4. User's transaction is mined and the user buys `1000 USDC` for very high price
5. Attacker sells all the `USDC` tokens they bought and makes profit

Recommended Mitigation: Define the `maxInputAmount` parameter for function similar to how `swapExactInput` has `minOutputAmount`. Then before calling `_swap` validate that the `inputAmount` is not higher than `maxInputAmount`.

```
    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 TSwapPool__InputTooHigh(inputAmount, maxInputAmount);
+       }

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

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

Description: The `sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. Users indicate how many pool tokens they are willing to sell in the `poolTokenAmount` parameter. However, the function currently miscalculates the swapped amount.

This is due to the fact that the function calls `swapExactOutput` with `outputToken` set to `poolToken`. However, the function should call `swapExactInput` with `inputToken` set to `poolToken`. Because user specifies the input amount not the output amount.

, the `swapExactOutput` is called with `outputToken` set to `poolToken`. This means that the user is buying the defined amount of weth tokens instead of selling the pool tokens. This means that the user will receive incorrect amount of tokens.

Impact: Users will swap the wrong amount of tokens, which is a critical issue.

Proof of Concept:

1. Liquidity provider deposits `100'000 USDC` and `100 WETH` to the pool
2. `1 WETH` is worth `1000 USDC`
3. User calls `TSwapPool::sellPoolTokens` with `poolTokenAmount` set to `50 USDC`
4. Functions call `getInputAmountBasedOnOutput` to calculate the `inputAmount`
   i. `outputReserves` is `100 WETH`
   ii. `inputReserves` is `100'000 USDC`
   iii. `outputAmount` is `50 USDC`
   iv. I am going to use a fixed `getInputAmountBasedOnOutput` function for clear visibility
   v. `((inputReserves * outputAmount) * 1000) / ((outputReserves - outputAmount) * 997)`
   vi. `((100'000 * 50) * 1000) / ((100 - 50) * 997)` (with fixed formula)
   vii. `~= 100,300.90 USDC`
5. Users wanted to sell `50 USDC` but instead they sold `100,300.90 USDC` worth of pool tokens.

Paste this test in `test/unit/TSwapPool.t.sol` directory. Note that test uses old contract formula which uses `10000` instead of `1000`.

```
        function test_audit_sellPoolTokensSellsIncorrectAmount() public {
        poolToken.mint(user, 100_000_000 ether);
        weth.mint(user, 100_000_000 ether);
        poolToken.mint(liquidityProvider, 1_000_000 ether);
        weth.mint(liquidityProvider, 1_000_000 ether);

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100 ether);
        poolToken.approve(address(pool), 100_000 ether);
        pool.deposit(
            100 ether,
            100 ether,
            100_000 ether,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        uint256 sellAmount = 50 ether;
        uint256 startingUserPoolTokenBalance = poolToken.balanceOf(user);

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

        uint256 endingUserPoolTokenBalance = poolToken.balanceOf(user);
        uint256 poolTokenBalanceDiff = startingUserPoolTokenBalance -
            endingUserPoolTokenBalance;
        uint256 actualSoldAmount = ((100_000 ether * 50 ether) *
            uint256(10000)) / ((100 ether - 50 ether) * uint256(997));

        assertNotEq(actualSoldAmount, sellAmount);
        assertEq(poolTokenBalanceDiff, actualSoldAmount);
    }
```

Recommended Mitigation: Use `swapExactInput` instead of `swapExactOutput` to sell pool tokens. Note that this would also require changing the `sellPoolTokens` function to accept `minWethToReceive` which should be passed to `swapExactInput` as `minOutputAmount`.

```diff
    function sellPoolTokens(
        uint256 poolTokenAmount
    ) external returns (uint256 wethAmount) {
        return
-           swapExactOutput(
-               i_poolToken,
-               i_wethToken,
-               poolTokenAmount,
-               uint64(block.timestamp)
-           );
+           swapExactInput(
+               i_poolToken,
+               i_wethToken,
+               poolTokenAmount,
+               0.001 ether,
+               uint64(block.timestamp)
+           );
    }
```

[H-4] Transfering tokens for extra incentives in `TSwapPool::_swap` breaks the main protocol `x * y = k` invariant

Description: The protocol follows a strict invariant of `x * y = k`, where `x` is the amount of pool tokens in the pool, `y` is the amount of WETH tokens in the pool and `k` is constant product of the two balances. This means that whenever the balances change in the protocol, the ratio of the two balances must remain the same. However, this is broken due to the fact that the `_swap` function transfers `1e18` of tokens to the user if the swap count reaches `SWAP_COUNT_MAX`. Meaning that over time the protocol will be drained of tokens.

Additionally, the same amount `1e18` is used no matter what type of output token is used (pool token or WETH). This means that some users will get more value than others.

The invariant is broken in the following code:

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

Impact: A user could maliciously drain the protocol of funds by doing a lot of swaps and collecting the extra incentive given out by the protocol.

Proof of Concept:

1. A user swaps 10 times and collects the extra incentive of `1e18` tokens
2. That user continues to swap untill all the protocol funds are drained

POC code

Paste the following test into `test/unit/TSwapPool.t.sol` file

```
    function test_audit_ConstantProductInvariantBreaks() 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 swapCount = 10;
        uint256 outputWeth = 1e18;
        int256 startingWethAmount = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaWethAmount = -1 *
            int256(outputWeth) *
            int256(swapCount);

        vm.startPrank(user);
        poolToken.mint(user, 1000e18);
        poolToken.approve(address(pool), type(uint256).max);
        for (uint256 i = 0; i < swapCount; i++) {
            pool.swapExactOutput(
                poolToken,
                weth,
                outputWeth,
                uint64(block.timestamp)
            );
        }
        vm.stopPrank();

        int256 endingWethAmount = int256(weth.balanceOf(address(pool)));
        int256 actualDeltaWethAmount = endingWethAmount - startingWethAmount;
        assertEq(
            actualDeltaWethAmount,
            expectedDeltaWethAmount,
            "WETH amount did not change as expected"
        );
    }
```

Recommended Mitigation: There are few options to mitigate this issue:

1. Remove the transfer of tokens from the pool reserves. This removes the possibility of draining the protocol of funds, but also removes the incentive for users to swap.
2. Mint or transfer another ERC20 token created only for incentives to users. This means that the pool will have the correct value and users will continue to get incentive to swap.

Medium

[M-1] `TSwapPool::deposit` does not check `deadline` parameter, causing the transaction to complete even after the deadline

Description: In `TSwapPool::deposit` , the `deadline` parameter should limit the transaction execution time according to the documentation: "The deadline for the transaction to be completed by". However, the `deadline` parameter is not checked. This means that the transaction can be executed even after the deadline.

Impact: Transactions could be executed when market conditions are unfavorable to deposit, even with a deadline set.

Proof of Concept:

1 User calls `TSwapPool::deposit` with `maximumPoolTokensToDeposit` amount and `deadline` set to `block.timestamp` . 2 Miner sees the transaction and waits until the price is more favorable for them. 3 Miner executes the transaction, which deposits the `maximumPoolTokensToDeposit` amount of users money.

Recommended Mitigation: Use already defined modifier `revertIfDeadlinePassed` to validate the `deadline` parameter.

```
    modifier revertIfDeadlinePassed(uint256 deadline) {
        if (deadline < block.timestamp) {
            revert TSwapPool__DeadlinePassed();
        }
        _;
    }



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

Low

[L-1] Incorrect parameters order for `TSwapPool::LiquidityAdded` event in `TSwapPool::_addLiquidityMintAndTransfer` causes incorrect data emitted in event logs

Description: In `TSwapPool::_addLiquidityMintAndTransfer`, the `TSwapPool::LiquidityAdded` event is emitted with incorrect arguments order.

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

Proof of Concept: Second parameter is `wethDeposited`, which is the third parameter in the event. And third parameter is `poolTokensMinted`, which is the second parameter in the event. This might cause confusion and incorrect data in subgraph.

Recommended Mitigation: Change the order of the arguments in the event.

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

[L-2] Default value (equal to `0`) is returned by `TSwapPool::swapExactInput` resulting in incorrect return value

Description: In `TSwapPool::swapExactInput` should return the amount of output tokens user received. However, the function is never assigned a value, so it returns the default value, which is `0`.

Impact: The function always returns `0`. This might cause confusion and unexpected bugs when calling the function.

Proof of Concept: The function will always return `0` instead of the amount of output tokens user received.

Recommended Mitigation: Return the output amount.

```diff
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

-        uint256 outputAmount = getOutputAmountBasedOnInput(
+        output = getOutputAmountBasedOnInput(
            inputAmount,
            inputReserves,
            outputReserves
        );

-       if (outputAmount < minOutputAmount) {
-           revert TSwapPool__OutputTooLow(outputAmount, minOutputAmount);
-       }
+       if (output < minOutputAmount) {
+           revert TSwapPool__OutputTooLow(output, minOutputAmount);
+       }

-       _swap(inputToken, inputAmount, outputToken, outputAmount);
+       _swap(inputToken, inputAmount, outputToken, output);
    }
```

Informational

[I-1] `PoolFactory::constructor` paramerter `wethToken` is not checked, so `i_wethToken` can be set to zero address

Description: In `PoolFactory::constructor` , the `wethToken` parameter is used to assign value to `i_wethToken` , but the parameter is not checked to ensure it is not the zero address. This means that `i_wethToken` can be set to the zero address, which can cause a `PoolFactory::createPool` call to fail.

Recommended Mitigation: Add a check to ensure that `wethToken` is not the zero address. Use modifier for reusability.

Modifier example:

```
    modifier notZeroAddress(address _address) {
        if (_address == address(0)) {
            revert PoolFactory__ZeroAddress();
        }
        _;
    }
```

[I-2] `PoolFactory::createPool` paramerter `tokenAddress` is not checked for zero address, which will revert when calling `ERC20::name`

Description: In `PoolFactory::createPool` , the `tokenAddress` parameter is used to create pool. The parameter is not checked to ensure it is not the zero address. This means that `tokenAddress` can be set to the zero address, which will cause a revert when calling `ERC20::name` .

Recommended Mitigation: Add a check to ensure that `tokenAddress` is not the zero address. Use modifier for reusability.

Modifier example:

```
    modifier notZeroAddress(address _address) {
        if (_address == address(0)) {
            revert PoolFactory__ZeroAddress();
        }
        _;
    }
```

[I-3] `TSwapPool::constructor` does not check for empty strings

Description: In `TSwapPool::constructor` , the `liquidityTokenName` and `liquidityTokenSymbol` are used to create `ERC20` token. The parameters are not checked to ensure they are not empty strings. This means that `liquidityTokenName` and `liquidityTokenSymbol` can be set to empty strings, which can cause confusion.

Recommended Mitigation: Add a check to ensure that `liquidityTokenName` and `liquidityTokenSymbol` are not empty strings. Use modifier for reusability.

Modifier example:

```
    modifier notEmptyString(string memory str) {
        if (bytes(str).length == 0) {
            revert TSwapPool__EmptyString();
        }
        _;
    }
```

[I-4] `PoolFactory::createPool` uses pool token name for LP token symbol

Description: In `PoolFactory::createPool` , the pool token name is used to concatenate with preffix `ts` to create LP token symbol. This means that the LP token symbol will be almost the same as the pool token name, which can cause confusion. Also names might have spaces, which might also be confusing.

Recommended Mitigation: Use pool token symbol instead of name to create LP token symbol to avoid confusion.

[I-5] `TSwapPool` state variables are not in order

Description: In `TSwapPool` , the state variables are immutable, constant, and then non-constant but they are not in order. This means that the state variables are not easy to read.

Recommended Mitigation: Order the state variables so that they are easy to read. Firstly, constant state variables, then immutable state variables, and then non-constant state variables.

[I-6] `TSwapPool::constructor` does not check for zero addresses

Description: In `TSwapPool::constructor` , the `poolToken` and `wethToken` are used to save the addresses to state variables. The parameters are not checked to ensure they are not the zero address. This means that `poolToken` and `wethToken` can be set to the zero address or empty strings, which can cause reverts when calling `ERC20` functions.

Recommended Mitigation: Add a check to ensure that `poolToken` and `wethToken` are not the zero address or empty strings. Use modifier for reusability.

Modifier example:

```
    modifier notZeroAddress(address addr) {
        if (addr == address(0)) {
            revert TSwapPool__ZeroAddress();
        }
        _;
    }
```

[I-7] `TSwapPool::deposit` external call before changing return value in `else` condition block

Description: In `TSwapPool::deposit` , the `else` condition block calls `_addLiquidityMintAndTransfer` function which is an external call. The return value is changed after the external call. This does not follow Checks-Effects-Interactions pattern.

Recommended Mitigation: Change the return value before the external call.

[I-8] `getOutputAmountBasedOnInput` , `getInputAmountBasedOnOutput` , `swapExactInput` have no natspec comments

Description: In `getOutputAmountBasedOnInput` , `getInputAmountBasedOnOutput` , `swapExactInput` , there are no natspec comments. This means that the functions are not easy to understand.

Recommended Mitigation: Add natspec comments to the functions. Explain the parameters and return values.

[I-9] `TSwapPool::getOutputAmountBasedOnInput` contains "magic number", which are not constants

Description: In `TSwapPool::getOutputAmountBasedOnInput` , the `997` and `1000` is used as a "magic numbers" to calculate the output amount. These "magic numbers" are not a constant. This means that these values can cause unexpected bugs as they are not easy to read and understand.

Recommended Mitigation: Add constants for the `997` and `1000` values.

[I-10] `TSwapPool::getInputAmountBasedOnOutput` contains "magic number", which are not constants

Description: In `TSwapPool::getInputAmountBasedOnOutput` , the `997` and `10000` is used as a "magic numbers" to calculate the input amount. These "magic numbers" are not a constant. This means that these values can cause unexpected bugs as they are not easy to read and understand.

Recommended Mitigation: Add constants for the `997` and `10000` values.

[I-11] `TSwapPool::swapExactInput` uses tokens `inputToken` and `outputToken` without checking if they are the same or zero addresses

Description: In `TSwapPool::swapExactInput` the `inputToken` and `outputToken` are used to calculate the output amount. The tokens are not checked to ensure they are not the same or zero addresses. This means that the function can be called with the same token as input and output, which can cause unexpected bugs. Or the function can be called with zero address, which can cause reverts when calling `ERC20` functions.

Recommended Mitigation: Add a check to ensure that `inputToken` and `outputToken` are not the same or zero addresses. Use modifiers for reusability.

Modifier example:

```
    modifier notZeroAddress(address addr) {
        if (addr == address(0)) {
            revert TSwapPool__ZeroAddress();
        }
        _;
    }


    modifier notSameAddress(address addr1, address addr2) {
        if (addr1 == addr2) {
            revert TSwapPool__SameAddress();
        }
        _;
    }
```

[I-12] `TSwapPool::swapExactOutput` is missing nat spec comment for `deadline` parameter

Description: In `TSwapPool::swapExactOutput` , the `deadline` parameter is missing natspec comment. This means that the function is not easy to understand.

Recommended Mitigation: Add natspec comment for the `deadline` parameter.

[I-13] `TSwapPool::swapExactOutput` uses tokens `inputToken` and `outputToken` without checking if they are the same or zero addresses

Description: In `TSwapPool::swapExactOutput` the `inputToken` and `outputToken` are used to calculate the input amount. The tokens are not checked to ensure they are not the same or zero addresses. This means that the function can be called with the same token as input and output, which can cause unexpected bugs. Or the function can be called with zero address, which can cause reverts when calling `ERC20` functions.

Recommended Mitigation: Add a check to ensure that `inputToken` and `outputToken` are not the same or zero addresses. Use modifiers for reusability.

Modifier example:

```
    modifier notZeroAddress(address addr) {
        if (addr == address(0)) {
            revert TSwapPool__ZeroAddress();
        }
        _;
    }


    modifier notSameAddress(address addr1, address addr2) {
        if (addr1 == addr2) {
            revert TSwapPool__SameAddress();
        }
        _;
    }
```

[I-14] `TSwapPool::_swap` uses "magic number" `1_000_000_000_000_000_000`

Description: In `TSwapPool::_swap` , the `1_000_000_000_000_000_000` is used as a "magic number" to calculate the amount. This "magic number" is not a constant. This means that this value can cause unexpected bugs as it is not easy to read and understand.

Recommended Mitigation: Add a constant for the `1_000_000_000_000_000_000` value. Use `1e18` instead or `1 ether` (recommended).

[I-15] `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` use "magic number"

Description: In `TSwapPool::getPriceOfOneWethInPoolTokens` and `TSwapPool::getPriceOfOnePoolTokenInWeth` , the `1e18` is used as a "magic number" to calculate the amount. This "magic number" is not a constant. This means that this value can cause unexpected bugs as it is not easy to read and understand.

Recommended Mitigation: Add a constant for the `1e18` value. Use `1 ether` instead of `1e18` (recommended).

Gas Optimization

[G-1] `PoolFactory::PoolFactory__PoolDoesNotExist` error is defined but not used anywhere

Description: In `PoolFactory` , the `PoolFactory__PoolDoesNotExist` error is defined but not used anywhere.

Recommended Mitigation: Remove the error definition to save contract deployment gas.

[G-2] `TSwapPool::TSwapPool__WethDepositAmountTooLow` reverts with `MINIMUM_WETH_LIQUIDITY` as first argument, which is constant value in contract

Description: In `TSwapPool::TSwapPool__WethDepositAmountTooLow`, the `MINIMUM_WETH_LIQUIDITY` is used as first argument to revert. The `MINIMUM_WETH_LIQUIDITY` is a constant value in the contract. This means that the revert message will always be the same and anyone can find out the value of `MINIMUM_WETH_LIQUIDITY` by reading the contract. This just costs gas and does not provide any value.

Recommended Mitigation: Remove the `MINIMUM_WETH_LIQUIDITY` from the revert message.

[G-3] `TSwapPool::poolTokenReserves` is assigned to pool token balance but it is not used anywhere

Description: In `TSwapPool::deposit`, the `poolTokenReserves` is assigned to pool token balance but it is not used anywhere. This means that the assignment is not needed. And only costs gas to execute external call to `ERC20::balanceOf`.

Recommended Mitigation: Remove the assignment to `poolTokenReserves`.

[G-4] `TSwapPool::swapExactInput` is set to be `public` but it is not used anywhere internally

Description: In `TSwapPool::swapExactInput`, the function's visibility is set to be `public` but it is not used anywhere internally. This means that the function can be set to be `external` to save gas.

Recommended Mitigation: Set the function's visibility to be `external`.

[G-5] `TSwapPool::_swap` calls `safeTransfer` for `outputToken` twice, which is not needed

Description: In `TSwapPool::_swap`, the `safeTransfer` is called for `outputToken` twice. First if swap count exceeds `MAX_SWAP_COUNT`, and second at the end of the function. This costs gas and is not needed.

```
    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);
```

Recommended Mitigation: Cache the total output amount and call `safeTransfer` for `outputToken` only once at the end of the function.

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