Protocol Audit Report

Prepared by: Sahil Kaushik

Table of Contents

* [I-3] `TSwapPool::constructor()` Lacks Zero Address Check
* [I-4] `MINIMUM_WETH_LIQUIDITY` Constant is Being Emitted During the Event in the `TSwapPool::deposit` Function
* [I-5] `poolTokenReserves` Are Not Being Used in `TSwapPool::deposit()` Function
* [I-6] Magic Numbers in `TSwapPool`
* [I-7] `TSwapPool::swapExactInput` & `TSwapPool::swapExactOutput` Provide No Natspec to Read and Understand the Functions
* [I-8] `TSwapPool::swapExactInput` Returns No Output, but a Return is Still Defined in the Function, Increasing Gas Cost

Protocol Summary

This contract allows user to enter a Raffle and winner can mint a random puppy nft.

Disclaimer

I made 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

Likelihood Impact	High	Medium	Low
High	H	H/M	M
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 the following commit hash:

```
1ec3c30253423eb4199827f59cf564cc575b46db
```

Scope

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

Roles

- Owner: Is the only one who should be able to set and access the password.

For this contract, only the owner should be able to interact with the contract. # Executive Summary ## Issues found

Severity	Number of issues found
High	5
Medium	0
Low	2
Info	8
Gas Optimizations	0
Total	15

Findings

High

[H-1] `TSwapPool::deposit()` Function Does Not Include a Deadline Check as Defined in the Natspec, Resulting in Deposition of Funds Even After the Deadline

Description: The `deadline` parameter defined in the `TSwapPool::deposit()` function is not being used to check the deadline time of the deposit. Therefore, users can deposit funds even after the deadline time period, resulting in breaking the protocol's deposit function as defined by the natspec.

Impact: Users can deposit funds even after the deadline.

Proof of Concept: In the `TSwapPool.sol`:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    @> uint64 deadline   // missing check
)
    external
    revertIfZero(wethToDeposit)
    returns (uint256 liquidityTokensToMint)
{
    if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
        // audit-Informational: MINIMUM_WETH_LIQUIDITY is a constant,
        ↪  emitting unnecessary

        revert TSwapPool__WethDepositAmountTooLow(
            MINIMUM_WETH_LIQUIDITY,
            wethToDeposit
        );
    }
    if (totalLiquidityTokenSupply() > 0) {
        uint256 wethReserves = i_wethToken.balanceOf(address(this));
        // audit not in use
        uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
        // Our invariant says weth, poolTokens, and liquidity tokens must
        ↪  always have the same ratio after the
        // initial deposit
        // poolTokens / constant(k) = weth
        // weth / constant(k) = liquidityTokens
```

```
        // aka...
        // weth / poolTokens = constant(k)
        // To make sure this holds, we can make sure the new balance will
        ↪  match the old balance
        // (wethReserves + wethToDeposit) / (poolTokenReserves +
        ↪  poolTokensToDeposit) = constant(k)
        // (wethReserves + wethToDeposit) / (poolTokenReserves +
        ↪  poolTokensToDeposit) =
        // (wethReserves / poolTokenReserves)
        //
        // So we can do some elementary math now to figure out
        ↪  poolTokensToDeposit...
        // (wethReserves + wethToDeposit) / poolTokensToDeposit =
        ↪  wethReserves
        // (wethReserves + wethToDeposit)  = wethReserves *
        ↪  poolTokensToDeposit
        // (wethReserves + wethToDeposit) / wethReserves  =
        ↪  poolTokensToDeposit
        uint256 poolTokensToDeposit = getPoolTokensToDepositBasedOnWeth(
            wethToDeposit
        );
        if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
            revert TSwapPool__MaxPoolTokenDepositTooHigh(
                maximumPoolTokensToDeposit,
                poolTokensToDeposit
            );
        }

        // We do the same thing for liquidity tokens. Similar math.
        liquidityTokensToMint =
            (wethToDeposit * totalLiquidityTokenSupply()) /
            wethReserves;
        if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
            revert TSwapPool__MinLiquidityTokensToMintTooLow(
                minimumLiquidityTokensToMint,
                liquidityTokensToMint
            );
        }
        _addLiquidityMintAndTransfer(
            wethToDeposit,
            poolTokensToDeposit,
            liquidityTokensToMint
        );
```

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

        // audit informational not follows CEI
        liquidityTokensToMint = wethToDeposit;
    }
}
```

Proof of Code

Place this in TSwapPool.t.sol:

```
function testDepositDeadline() public {
    vm.startPrank(liquidityProvider);

    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);

    vm.expectRevert("TSwapPool__DeadlinePassed");
    pool.deposit(
        100e18,
        100e18,
        100e18,
        uint64(block.timestamp - 1)
    );
}
```

Output:

```
Ran 1 test for test/unit/TSwapPool.t.sol:TSwapPoolTest
[FAIL: next call did not revert as expected] testDepositDeadline() (gas: 191547)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.07ms (182.64µ
```

Recommended Mitigation:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    uint64 deadline
)
    external
    revertIfZero(wethToDeposit)
    returns (uint256 liquidityTokensToMint)
{
+   revertIfDeadlinePassed(deadline);
    if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
        // audit-Informational: MINIMUM_WETH_LIQUIDITY is a constant,
↪   emitting unnecessary

        revert TSwapPool__WethDepositAmountTooLow(
            MINIMUM_WETH_LIQUIDITY,
            wethToDeposit
        );
    }
    if (totalLiquidityTokenSupply() > 0) {
        uint256 wethReserves = i_wethToken.balanceOf(address(this));
        // audit not in use
        uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
        // Our invariant says weth, poolTokens, and liquidity tokens must
↪   always have the same ratio after the
        // initial deposit
        // poolTokens / constant(k) = weth
        // weth / constant(k) = liquidityTokens
        // aka...
        // weth / poolTokens = constant(k)
        // To make sure this holds, we can make sure the new balance will
↪   match the old balance
        // (wethReserves + wethToDeposit) / (poolTokenReserves +
↪   poolTokensToDeposit) = constant(k)
        // (wethReserves + wethToDeposit) / (poolTokenReserves +
↪   poolTokensToDeposit) =
        // (wethReserves / poolTokenReserves)
        //
        // So we can do some elementary math now to figure out
↪   poolTokensToDeposit...
        // (wethReserves + wethToDeposit) / poolTokensToDeposit =
↪   wethReserves
```

```
        // (wethReserves + wethToDeposit)  = wethReserves *
↪   poolTokensToDeposit
        // (wethReserves + wethToDeposit) / wethReserves  =
↪   poolTokensToDeposit
        uint256 poolTokensToDeposit = getPoolTokensToDepositBasedOnWeth(
            wethToDeposit
        );
        if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
            revert TSwapPool__MaxPoolTokenDepositTooHigh(
                maximumPoolTokensToDeposit,
                poolTokensToDeposit
            );
        }

        // We do the same thing for liquidity tokens. Similar math.
        liquidityTokensToMint =
            (wethToDeposit * totalLiquidityTokenSupply()) /
            wethReserves;
        if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
            revert TSwapPool__MinLiquidityTokensToMintTooLow(
                minimumLiquidityTokensToMint,
                liquidityTokensToMint
            );
        }
        _addLiquidityMintAndTransfer(
            wethToDeposit,
            poolTokensToDeposit,
            liquidityTokensToMint
        );
    } else {
        // This will be the "initial" funding of the protocol. We are
↪   starting from blank here!
        // We just have them send the tokens in, and we mint liquidity
↪   tokens based on the weth
        _addLiquidityMintAndTransfer(
            wethToDeposit,
            maximumPoolTokensToDeposit,
            wethToDeposit
        );

        // audit informational not follows CEI
        liquidityTokensToMint = wethToDeposit;
    }
```

```
}
```

[H-2] Incorrect Fee Calculation in `TSwapPool::getInputAmountBasedOnOutput` Causes Protocol to Take Too Many Tokens From Users, Resulting in Lost Fees

Description: While calculating the input amount in `TSwapPool::getInputAmountBasedOnOutput()`, the function multiplies the amount by 10000 instead of 1000, resulting in users having to pay ten times more tokens than needed to get the required amount of WETH tokens.

Impact: Users have to pay ten times more for swapping tokens.

Proof of Concept: In `TSwapPool::getInputAmountBasedOnOutput()`:

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

Proof of Code

Place this in test suite in `TSwapPool.t.sol`:

```
function testInputBasedOnOutput() public {
    uint256 inputReserves = 100_000;
    uint256 outputReserves = 100_000;
    uint256 outputAmount = 1000;

    uint256 correctInput = ((inputReserves * outputAmount) * 1000) /
        ((outputReserves - outputAmount) * 997);

    uint256 poolInput = pool.getInputAmountBasedOnOutput(
```

```
        outputAmount,
        inputReserves,
        outputReserves
    );

    // uint256 poolInput = ((inputReserves * outputAmount) * 10000) /
    //     ((outputReserves - outputAmount) * 997);
    console.log("Correct Input:", correctInput);
    console.log("Pool Input:", poolInput);
    assertEq(correctInput, poolInput);
}
```

Output:

```
Ran 1 test for test/unit/TSwapPool.t.sol:TSwapPoolTest
[FAIL] testInputBasedOnOutput() (gas: 29816)
Logs:
  Error: a == b not satisfied [uint]
        Left: 1013
       Right: 10131

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 2.23ms (244.10µ
```

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);
+       ((inputReserves * outputAmount) * 1000) /
+       ((outputReserves - outputAmount) * 997);
}
```

[H-3] Lack of Slippage Protection in `TSwapPool::swapExactOutput` Causes Users to Potentially Receive Way Fewer Tokens

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

Impact: If market conditions change before the transaction processes, the user could get a much worse swap.

Proof of Concept:

1. The price of 1 WETH right now is 1,000 USDC.
2. User inputs a `swapExactOutput` looking for 1 WETH:
    - inputToken = USDC
    - outputToken = WETH
    - outputAmount = 1
    - deadline = whatever
3. The function does not offer a maxInput amount.
4. As the transaction is pending in the mempool, the market changes! The price moves significantly—1 WETH is now 10,000 USDC (10x more than the user expected).
5. The transaction completes, but the user sent the protocol 10,000 USDC instead of the expected 1,000 USDC.

Recommended Mitigation:

```
function swapExactOutput(
    IERC20 inputToken,
+   uint256 maxInputAmount,
    .
    .
    .
    inputAmount = getInputAmountBasedOnOutput(outputAmount, inputReserves,
↪   outputReserves);
+   if(inputAmount > maxInputAmount){
+       revert();
+   }
    _swap(inputToken, inputAmount, outputToken, outputAmount);
)
```

[H-4] `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 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 `swapExactOutput` function is called, whereas the `swapExactInput` function should be called instead. Users specify the exact amount of input tokens, not output.

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

Proof of Concept:

1. A user intends to sell a fixed amount of pool tokens in exchange for WETH by calling `sellPoolTokens(poolTokenAmount)`. Here, poolTokenAmount represents the exact input amount the user is willing to sell.
2. The user approves the pool to spend exactly poolTokenAmount pool tokens, indicating a hard upper bound on the input amount.
3. Internally, sellPoolTokens calls swapExactOutput, which is designed for swaps where the output amount is fixed and the input amount is computed dynamically.
4. Because swapExactOutput treats the output amount as fixed, the pool calculates the required input amount at execution time based on current pool reserves.
5. As a result, the computed input amount may exceed poolTokenAmount, causing the pool to attempt to transfer more input tokens than the user specified.

This behavior is observable in execution: the pool attempts to pull a larger amount of pool tokens than the user approved, demonstrating that the function is executing exact-output semantics, despite the user specifying an exact input amount.

This proves that sellPoolTokens incorrectly uses swapExactOutput, and that swapExactInput is the correct function to call for this user flow.

```
function sellPoolTokens(
    uint256 poolTokenAmount
) external returns (uint256 wethAmount) {
    return
        @> swapExactOutput(
            i_poolToken,
```

```
            i_wethToken,
            poolTokenAmount,
            uint64(block.timestamp)
        );
}
```

Recommended Mitigation:

Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (i.e., `minWethToReceive` to be passed to `swapExactInput`).

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

[H-5] In `TSwapPool::_swap`, the Extra Tokens Given to Users After Every `swapCount` Breaks the Protocol Invariant of `x * y = k`

Description: The protocol follows a strict invariant of `x * y = k`, where: - x: The balance of the pool token - y: The balance of WETH - k: The constant product of the two balances

This means that whenever the balances change in the protocol, the ratio between the two amounts should remain constant, hence the k. However, this is broken due to the extra incentive in the `_swap` function, meaning that over time, the protocol funds will be drained.

The following block of code is responsible for the issue:

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

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

Most simply put, the protocol's core invariant is broken.

Proof of Concept:

1. A user swaps 10 times and collects the extra incentive of `1_000_000_000_000_000_000` tokens.
2. That user continues to swap until all the protocol funds are drained.

Proof of Code

Place the following into `TSwapPool.t.sol`:

```
function testInvariantBroken() public {
    vm.startPrank(liquidityProvider);
    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);
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));

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

```
    pool.swapExactOutput(poolToken, weth, outputWeth,
↪   uint64(block.timestamp));
    vm.stopPrank();

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

Recommended Mitigation:

Remove the extra incentive mechanism. If you want to keep this in, you should account for the change in the `x * y = k` protocol invariant, or you should set aside tokens in the same way you do with fees.

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

Medium

Low

[L-1] Mismatched Order of Parameters in the `LiquidityAdded` Event in `TSwapPool::_addLiquidityMintAndTransfer()` Function

Description: The `poolTokensToDeposit` and `wethDeposit` parameters are mismatched to each other in the `LiquidityAdded` event and can lead to false on-chain data when working off-chain.

Impact: Leads to incorrect information about the protocol off-chain.

Recommended Mitigation:

```
function _addLiquidityMintAndTransfer(
    uint256 wethToDeposit,
    uint256 poolTokensToDeposit,
    uint256 liquidityTokensToMint
) private {
    _mint(msg.sender, liquidityTokensToMint);
    // audit-Low - Ordering of event emissions incorrect, should be `emit
↪   LiquidityAdded(msg.sender, wethToDeposit, poolTokensToDeposit)`

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

    // Interactions
    i_wethToken.safeTransferFrom(msg.sender, address(this), wethToDeposit);
    i_poolToken.safeTransferFrom(
        msg.sender,
        address(this),
        poolTokensToDeposit
    );
}
```

[L-2] `TSwapPool::sellPoolTokens()` Always Returns 0 as the Output `wethAmount`, Resulting in Misinformation About the Amount of WETH Tokens Received

Description: Since the `TSwapPool::swapExactOutput()` does not return anything on function call, the `sellPoolTokens` always returns an empty value, which in this case will be 0 due to the uint256 return expression.

Impact: It always gives incorrect information about the amount of WETH received.

Recommended Mitigation:

Either include a return statement which returns the wethAmount, or remove the return from `sellPoolTokens`.

```
function sellPoolTokens(
    uint256 poolTokenAmount
) external
-   returns (uint256 wethAmount) {
    return
        swapExactOutput(
```

```
            i_poolToken,
            i_wethToken,
            poolTokenAmount,
            uint64(block.timestamp)
        );
}
```

Informational

[I-1] Error `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` is Not Used, Leading to Increased Deployment Gas Cost

Description: Error `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` is being initialized and is not in use. This occupies extra storage slots for storing the error bytecode and ultimately increases the overall deployment cost (gas cost) of the contract.

Impact: Increased deployment gas cost.

Recommended Mitigation:

Remove the error:

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

[I-2] `PoolFactory::createPool` Should Use `.symbol()` Instead of `.name()` for `liquidityTokenSymbol`

Recommended Mitigation:

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

[I-3] `TSwapPool::constructor()` Lacks Zero Address Check

Recommended Mitigation:

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

[I-4] `MINIMUM_WETH_LIQUIDITY` Constant is Being Emitted During the Event in the `TSwapPool::deposit` Function

Description: There is no need to emit constants as they only contribute to increased gas costs.

Impact: Increased gas cost.

Proof of Code: In `TSwapPool.sol`:

```
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));
        // audit not in use
        uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
        // Our invariant says weth, poolTokens, and liquidity tokens must
         ↪  always have the same ratio after the
        // initial deposit
        // poolTokens / constant(k) = weth
        // weth / constant(k) = liquidityTokens
        // aka...
        // weth / poolTokens = constant(k)
        // To make sure this holds, we can make sure the new balance will
         ↪  match the old balance
        // (wethReserves + wethToDeposit) / (poolTokenReserves +
         ↪  poolTokensToDeposit) = constant(k)
        // (wethReserves + wethToDeposit) / (poolTokenReserves +
         ↪  poolTokensToDeposit) =
        // (wethReserves / poolTokenReserves)
        //
        // So we can do some elementary math now to figure out
         ↪  poolTokensToDeposit...
        // (wethReserves + wethToDeposit) / poolTokensToDeposit =
         ↪  wethReserves
        // (wethReserves + wethToDeposit)  = wethReserves *
         ↪  poolTokensToDeposit
        // (wethReserves + wethToDeposit) / wethReserves  =
         ↪  poolTokensToDeposit
        uint256 poolTokensToDeposit = getPoolTokensToDepositBasedOnWeth(
            wethToDeposit
        );
        if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
            revert TSwapPool__MaxPoolTokenDepositTooHigh(
                maximumPoolTokensToDeposit,
                poolTokensToDeposit
            );
        }

        // We do the same thing for liquidity tokens. Similar math.
        liquidityTokensToMint =
            (wethToDeposit * totalLiquidityTokenSupply()) /
            wethReserves;
        if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
            revert TSwapPool__MinLiquidityTokensToMintTooLow(
```

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

        // audit informational not follows CEI
        liquidityTokensToMint = wethToDeposit;
    }
}
```

Recommended Mitigation:

Modify the event to not emit any constants.

[I-5] `poolTokenReserves` Are Not Being Used in `TSwapPool::deposit()` Function

Impact: Increased gas cost.

Recommended Mitigation:

Remove the `poolTokenReserves` variable:

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

[I-6] Magic Numbers in `TSwapPool`

Recommended Mitigation:

```
+ constant uint256 FEE_CONSTANT = 997;
+ constant uint256 FEE_ESTIMATION_CONSTANT = 1000;

function getOutputAmountBasedOnInput(
    uint256 inputAmount,
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(inputAmount)
    revertIfZero(outputReserves)
    returns (uint256 outputAmount)
{
    // x * y = k
    // numberOfWeth * numberOfPoolTokens = constant k
    // k must not change during a transaction (invariant)
    // with this math, we want to figure out how many PoolTokens to deposit
    // since weth * poolTokens = k, we can rearrange to get:
    // (currentWeth + wethToDeposit) * (currentPoolTokens +
↪   poolTokensToDeposit) = k
    // **************************
    // ****** MATH TIME!!! ******
    // **************************
    // FOIL it (or ChatGPT):
↪   [https://en.wikipedia.org/wiki/FOIL_method](https://en.wikipedia.org/wiki/FOIL_metho
    // (totalWethOfPool * totalPoolTokensOfPool) + (totalWethOfPool *
↪   poolTokensToDeposit) + (wethToDeposit *
    // totalPoolTokensOfPool) + (wethToDeposit * poolTokensToDeposit) = k
    // (totalWethOfPool * totalPoolTokensOfPool) + (wethToDeposit *
↪   totalPoolTokensOfPool) = k - (totalWethOfPool *
    // poolTokensToDeposit) - (wethToDeposit * poolTokensToDeposit)

    // audit magic numbers 997 and 1000 represent 0.3% fee
-   uint256 inputAmountMinusFee = inputAmount * 997;
+   uint256 inputAmountMinusFee = inputAmount * FEE_CONSTANT;
     uint256 numerator = inputAmountMinusFee * outputReserves;
-    uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
```

```
+     uint256 denominator = (inputReserves * FEE_ESTIMATION_CONSTANT) +
↪   inputAmountMinusFee;

    return numerator / denominator;
}
```

[I-7] `TSwapPool::swapExactInput` & `TSwapPool::swapExactOutput` Provide No Natspec to Read and Understand the Functions

Description: Natspec should be provided for these functions as they are important functions of the protocol. Without proper documentation, one cannot know whether they are working according to what the protocol intends during auditing.

[I-8] `TSwapPool::swapExactInput` Returns No Output, but a Return is Still Defined in the Function, Increasing Gas Cost

Recommended Mitigation:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    public
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
-   returns (
-       uint256 output
    )
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    uint256 outputAmount = getOutputAmountBasedOnInput(
```

```
        inputAmount,
        inputReserves,
        outputReserves
    );

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

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