E.A RESEARCH
SECURING SMART CONTRACTS

TSwap Protocol Audit Report

Version 1.0

E.A Research

August 23, 2024

TSwap Protocol Audit Report

Emmanuel Acho, PhD

August 23rd, 2024

Prepared by: Emmanuel Acho, PhD

Table of Contents

- High
  * [H-1] `TSwapPool::getInputAmountBasedOnOutput` has an incorrect fee calculation resulting in a 90.3% fee charge instead of 0.03%. This can result in fees being lost.
  * [H-2] `TSwapPool::swapExactOutput` has no slippage protection which will cause users to potentially receive less tokens if market conditions become unfavorable.
  * [H-3] `TSwapPool::sellPoolTokens` implements the wrong function call (`swapExactOutput`) leading to a mismatch between input and output tokens and users receiving an incorrect amount of tokens.
  * [H-4] `TSwapPool::_swap` transfers the `msg.sender` an extra output token after every 10 swaps which breaks the protocol invariant `x * y = k`
- Medium
- [M-1] `TSwapPool::deposit` does not use the `deadline` parameter/check which will cause transactions to complete even after the deadline
- Low
  * [L-1] `TSwapPool::LiquidityAdded` event has parameters listed out of order meaning wrong information is passed on to the user as the events emits incorrect information
- [L-2] **`public`** functions not used internally could be marked `external`
  * [L-3] `TSwapPool::swapExactInput` will result in an incorrect return value being passed on to the user.
- Informational
  * [I-1] `PoolFactory__PoolDoesNotExist` is not used and should preferably be removed.
  * [I-2] `PoolFactory::constructor(address wethToken)` does not have a zero check for the wethToken address.
  * [I-3] `PoolFactory::createPool()` uses `IERC20(tokenAddress).name()` instead of `.symbol()`
  * [I-4] `TSwapPool::Swap()` event has more than three parameters but only one event is indexed. See below for details and other instances.
  * [I-5] `TSwapPool::constructor` is lacking a zero address check for the `poolToken` and `wethToken` variables.
  * [I-6] `TSwapPool::deposit` does not use the `poolTokenReserves` variable. Consider removing it.

Protocol Summary

The TSWAP protocol is a decentralized token swap and liquidity management system deployed on the Ethereum blockchain. It allows users to trade between WETH (Wrapped Ether) and other ERC20 tokens through automated liquidity pools. Each liquidity pool maintains a constant product invariant (`x * y = k`) to facilitate token swaps while ensuring that the ratio of assets remains balanced.

The protocol consists of two primary smart contracts:

1. **TSwapPool**: This contract handles the core functionalities of adding and removing liquidity, as well as executing token swaps. Users can deposit WETH and other ERC20 tokens into the pool, receive liquidity tokens in return, and later redeem these liquidity tokens to withdraw their share of the pool's assets. The TSwapPool contract also supports direct token swaps between WETH and the pool's specific ERC20 token, ensuring a seamless trading experience.
2. **PoolFactory**: The PoolFactory contract is responsible for creating new TSwapPool instances. Each pool manages a specific token pair, with WETH being one half of the pair. The factory contract ensures that each token has a unique associated pool and provides functions to look up pools by their associated token addresses.

The TSWAP protocol aims to provide a secure, efficient, and user-friendly platform for decentralized trading and liquidity provision, leveraging the power of smart contracts to automate and streamline token exchanges.

Disclaimer

The E.A Research team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M

		Impact		
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

Scope

```
./src/
@> PoolFactory.sol
@> TSwapPool.sol
```

Roles

1. **Liquidity Providers (LPs):** Deposit WETH and another ERC20 token into the liquidity pools and receive liquidity tokens in return.
2. **Traders:** Swap one token for another using the liquidity provided in the TSWAP pools, paying a fee for each swap.
3. **Pool Creators:** Initialize new liquidity pools for specific ERC20 tokens paired with WETH using the PoolFactory contract.

Executive Summary

Many Thanks to the Cyfrin Updraft Team for this guided audit of the TSwap Protocol.

Overview

The TSWAP protocol is a decentralized token swap and liquidity provision system built using smart contracts on the Ethereum blockchain. It primarily consists of two smart contracts: `TSwapPool` and `PoolFactory`. The `TSwapPool` contract handles the liquidity management, swapping, and trading functionalities. It allows users to add and remove liquidity in the form of WETH and another ERC20 pool token, facilitating trading between these tokens while maintaining a constant product invariant (`x * y = k`). The `PoolFactory` contract allows for the creation of new `TSwapPool` instances, each managing a different token pair.

Audit Summary

The audit identified several critical and high-severity issues that could significantly impact the protocol's functionality and security. In total, 13 issues were found, categorized as follows:

- **High Severity**: 4
- **Medium Severity**: 1
- **Low Severity**: 2
- **Informational**: 6

High-Severity Issues

1. **Incorrect Fee Calculation in `getInputAmountBasedOnOutput`**:
   - The function charges a fee of 90.3% instead of the intended 0.3% due to a scaling error. This miscalculation could lead to substantial overcharging, negatively affecting user experience and the protocol's integrity.
2. **Lack of Slippage Protection in `swapExactOutput`**:
   - The absence of a maximum input amount exposes users to potential losses during price volatility. Without slippage protection, users could end up spending significantly more tokens than expected.
3. **Incorrect Function Call in `sellPoolTokens`**:
   - The function uses `swapExactOutput` instead of `swapExactInput`, leading to mismatches between input and output token amounts. This error can cause users to receive incorrect amounts of WETH, disrupting the intended functionality of the protocol.
4. **Protocol Invariant Violation due to Incentive Mechanism**:
   - Every 10 swaps, the `TSwapPool::_swap` function transfers an extra token to the `msg.sender`, breaking the protocol's invariant of `x * y = k`. This flaw could be exploited to drain the protocol's funds over time.

Medium-Severity Issue

- **Unused Deadline in `deposit` Function**:
  - The function accepts a `deadline` parameter but does not use it. This omission could lead to transactions being completed after the specified deadline, exposing users to potential front-running or unfavorable conditions.

Low-Severity Issues

1. **Incorrect Event Parameter Order in `LiquidityAdded`**:
   - The `LiquidityAdded` event emits parameters in the wrong order, potentially leading to confusion for off-chain services and users monitoring the protocol.
2. **Public Functions Not Marked as External**:
   - Several functions marked as **`public`** could be optimized by marking them as `external`, reducing gas costs and improving efficiency.

Informational Issues

- **Zero Address Checks Missing**: Both the `PoolFactory` and `TSwapPool` constructors lack checks for zero addresses, which could lead to unexpected behavior or security vulnerabilities.
- **Unindexed Event Parameters**: Several events with more than three parameters do not utilize indexing, which could hinder off-chain event monitoring and analysis.
- **Unused Variables**: The `deposit` function defines a `poolTokenReserves` variable that is never used, indicating possible inefficiencies or oversights in the code.

Recommendations

1. **Correct Fee Calculation**: Update the `getInputAmountBasedOnOutput` function to apply the correct fee, reducing the risk of user overcharging.
2. **Implement Slippage Protection**: Introduce a `maxInputAmount` parameter in the `swapExactOutput` function to provide a predictable ceiling on user spending.
3. **Fix Function Call in `sellPoolTokens`**: Replace the use of `swapExactOutput` with `swapExactInput` and pass an additional parameter for `minWethToReceive` to ensure users receive the correct amount of WETH.
4. **Remove Incentive Mechanism or Adjust to Maintain Invariant**: Eliminate the extra token transfer after every 10 swaps or find a method to maintain the protocol's invariant.

5. **Use Deadlines Appropriately**: Ensure that all functions with a `deadline` parameter enforce the deadline, reducing the risk of MEV attacks and user losses.
6. **Code Cleanup and Optimization**: Remove unused variables and implement zero address checks. Use external visibility for functions not called internally to save on gas costs.

Conclusion

The audit revealed critical vulnerabilities and areas for optimization within the TSWAP protocol. Addressing these issues is essential to ensure the security, efficiency, and reliability of the protocol. The TSWAP team should prioritize the high-severity findings and implement recommended mitigations to enhance the protocol's robustness and maintain user trust.

Issues found

Severity	Number of issues found
High	4
Medium	1
Low	2
Gas	0
Info	6
Total	13

Findings

High

[H-1] `TSwapPool::getInputAmountBasedOnOutput` has an incorrect fee calculation resulting in a 90.3% fee charge instead of 0.03%. This can result in fees being lost.

Description: The `getInputAmountBasedOnOutput` function returns the amount of tokens a user should deposit for a given amount of output tokens. In its current implemention, the user gets 997 output tokens for 10_000 input tokens instead of 997 for 1_000. The amount is scaled by 10_000 instead of 1_000.

Impact: The protocol is charging users more fees than expected. The user pays a 90.3% fee instead of 0.3%.

Proof of Concept: Please include the following test in `TSwapPool.t.sol` to illustrate potential for overcharging the user

Code

```
    function testBadExactOutput() public {
        // Let the liquidity provider provide liquidity
        uint256 initialLiquidty = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidty);
        poolToken.approve(address(pool), initialLiquidty);
        pool.deposit(
            initialLiquidty,
            initialLiquidty,
            initialLiquidty,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        // Let the user swap
        // Expected result with correct fee calculation: let's assume
           this should require ~10.134 units of poolToken
        // But with the bad fee logic, it will likely require more
        vm.startPrank(user);
        // poolToken.mint(user, 100e18);
        uint256 outputAmount = 1e18;
        uint256 expectedInputAmount = 10.134e18; // This is the
           expected input amount with the bad fee logic
        poolToken.approve(address(pool), expectedInputAmount);

        // Perform the swap
        uint256 requiredInput = pool.swapExactOutput(
            poolToken,
            weth,
            outputAmount,
            uint64(block.timestamp)
        );
        console.log("Required input: ", requiredInput);
        console.log("Expected input: ", expectedInputAmount);
        // Check that the input amount required is excessively high due
            to incorrect fee logic
        assertApproxEqAbs(
            requiredInput,
            expectedInputAmount,
            3e16,
            "Incorrect fee calculation should result in ~101343 input
               tokens required instead of the correct value."
        );

        vm.stopPrank();
    }
```

Recommended Mitigation

Code

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

        return
-               ((inputReserves * outputAmount) * 10000) /
+               ((inputReserves * outputAmount) * 1000) /
              ((outputReserves - outputAmount) * 997);
    }
```

[H-2] `TSwapPool::swapExactOutput` has no slippage protection which will cause users to potentially receive less tokens if market conditions become unfavorable.

Description: `swapExactOutput` is similar to `TSwapPool::swapExactInput`. The latter specifies a `minOutputAmount` preventing the user from receiving fewer tokens than expected. `swapExactOutput` should also specify a `maxInputAmount` to prevent the user from spending more tokens than they wish to if market conditions are unfavorable in order to receive the required amount of tokens, or fewer tokens for the intended spend.

Impact: Market conditions may change before the transaction is processed causing the user to get a much worse swap.

Proof of Concept: 1. The price of WETH is USD 1_000 2. The user calls `swapExactOutput` with 1 WETH as the output amount 3. The function does not offer a maxInput amount. 4. If the transaction is pending in the memepool, market conditions change and the price of ETH increases relative to what it was before it will now cost the user more poolTokens in order to get the required amount of WETH. 1. This will be the case if we go from 1 WETH = USD 1_000 to 1 WETH = USD 10_000. 2. The user will now have to transfer USD 10_000 for 1 WETH.

Please include the following test in `TSwapPool.t.sol` to illustrate the issue. Remember to mint additional poolTokens to the user before running the test to avoid an `ERC20InsufficientBalance` error.

Code

```
function testSwapExactOutputWithNoSlippageProtection()
            public
            liquidityProvided
      {
            vm.startPrank(user);
            uint256 initialPoolTokenBalance = poolToken.balanceOf(user);
            console.log("Initial pool token balance: ",
                initialPoolTokenBalance);
            uint256 outputAmount = 10e18;

            // User approves what they expect to be a reasonable amount
                based on current prices
            uint256 expectedInputAmount = 10e18;
            poolToken.approve(address(pool), type(uint256).max);
            vm.stopPrank();

            // Simulate a change in market conditions, making WETH more
                expensive
            vm.startPrank(liquidityProvider);
            pool.approve(address(pool), 50e18); // Approving 50 LP tokens
            pool.withdraw(50e18, 50e18, 50e18, uint64(block.timestamp));
            vm.stopPrank();

            // The user attempts to swap after the market conditions have
                changed
            vm.startPrank(user);
            uint256 requiredInput = pool.swapExactOutput(
                  poolToken,
                  weth,
                  outputAmount,
                  uint64(block.timestamp)
            );

            // Check if required input is greater than expected due to
                slippage
            console.log("Required input: ", requiredInput);
            console.log("Expected input: ", expectedInputAmount);
            assert(requiredInput > expectedInputAmount);
            vm.stopPrank();
      }
```

Recommended Mitigation Include a maxInput amount as a predictable ceiling for their spend on the protocol.

Code

```
          function swapExactOutput(
          IERC20 inputToken,
+         maxInputAmount
          IERC20 outputToken,
          uint256 outputAmount,
          uint64 deadline
      )
.
.
.
          inputAmount = getInputAmountBasedOnOutput(
              outputAmount,
              inputReserves,
              outputReserves
          );
+       if (inputAmount > maxInputAmount) {
+           revert()
+        }

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

[H-3] `TSwapPool::sellPoolTokens` implements the wrong function call (`swapExactOutput`) leading to a mismatch between input and output tokens and users receiving an incorrect amount of tokens.

Description: `sellPoolTokens` enables users to sell pool tokens for WETH. However, in its current implementation, it calls `TSwapPool::swapExactOutput` and the `poolTokenAmount` parameter is passed on as the `outputAmount` parameter. The function should call `TSwapPool::swapExactInput` instead, passing on the `poolTokenAmount` parameter as the `inputAmount` and `minWethToReceive` as the `minOutputAmount`.

Impact: The user will be inadvertently passing on the `poolTokenAmount` which is what the want to sell as the `outputAmount` which is what they want to receive in exchange. Hence, they will be swapping the wrong amount of tokens. This is a severe disruption to the intended functionality of the protocol.

Proof of Concept: Please include the following test in `TSwapPool.t.sol` to illustrate the issue. Remember to fix H1 above before running the test to avoid an `ERC20InsufficientBalance` error.

Code

```
function testSellPoolTokensIncorrectFunctionCall() public {
    // Step 1: Set up initial liquidity
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    // Step 2: Set up initial balances for user
    vm.startPrank(user);
    uint256 poolTokensToSell = 10e18; // The user wants to sell 10 pool
         tokens
    poolToken.approve(address(pool), type(uint256).max);

    // Calculate the expected WETH based on the current reserves
    uint256 expectedWethReceived = pool.getOutputAmountBasedOnInput(
        poolTokensToSell,
        poolToken.balanceOf(address(pool)),
        weth.balanceOf(address(pool))
    );
    vm.stopPrank();

    // Step 3: Call sellPoolTokens and capture the WETH received
    vm.startPrank(user);
    uint256 wethReceived = pool.sellPoolTokens(poolTokensToSell);
    uint256 actualPoolTokensSpent = pool.getInputAmountBasedOnOutput(
        wethReceived,
        poolToken.balanceOf(address(pool)),
        weth.balanceOf(address(pool))
    );
    vm.stopPrank();

    // Step 4: Assert that the received WETH is incorrect based on the
        pool's current reserves
    assert(wethReceived > expectedWethReceived);
    assert(actualPoolTokensSpent > poolTokensToSell);

    // Log the values for debugging
    console.log("Pool Tokens Sold: ", actualPoolTokensSpent);
    console.log("Expected Pool Tokens Sold: ", poolTokensToSell);
    console.log("WETH Received: ", wethReceived);
    console.log("Expected WETH Received: ", expectedWethReceived);
}
```

Recommended Mitigation The implementation should use `swapExactInput` instead of `swapExactOutput`. This will also updating `sellPoolTokens` to receive an additional parameter `minWethToReceive`.

Code

```
    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-4] `TSwapPool::_swap` transfers the `msg.sender` an extra output token after every 10 swaps which breaks the protocol invariant `x * y = k`

Description: The protocol makes sure that the product of the balance of the pool token:`x` and WETH:`y` remains constant:`k`. For this to be true, when balances change in the protocol, the ratio of weth to pool token balances should also remain constant. However, the extra transfer that is triggered every 10 swaps breaks the invariant and could eventually lead to protocol funds being drained over time. The following code snippet causes the issue

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 malicious user can take advantage of the extra incentive to drain the protocol by making multiple swaps. Also, the protocols invariant will be broken.

Proof of Concept: Please include the following test in `TSwapPool.t.sol` to illustrate the issue.

Code

```
function testInvariantBreaks() 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 = 1e18;

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        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); // -2000000000000000000
            != -1000000000000000000
    }
```

Recommended Mitigation It is best to remove the incentive mechanism as illustrated below or account for the change in `x * y = k`

Code

```
-         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

[M-1] `TSwapPool::deposit` does not use the `deadline` parameter/check which will cause transactions to complete even after the deadline

Description: The `deposit` function accepts a `deadline` parameter which sets the deadline for which the transaction should be completed by. However, the parameter is never used such that operations adding liquidity to the pool may be executed at unplanned/unexpected times which may be unfavorable (deposit rate) for the user.

Impact: Transactions could be sent at unfavorable times for the user even when a deadline is specified by them. This also creates an opportunity for MEVs (frontrunning)

Proof of Concept: The deadline parameter is unused

Code

```
        Warning (5667): Unused function parameter. Remove or comment
           out the variable name to silence this warning.
        --> src/TSwapPool.sol:125:9:
            |
        125 |         uint64 deadline
            |         ^^^^^^^^^^^^^^^
```

Recommended Mitigation

Code

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

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameters listed out of order meaning wrong information is passed on to the user as the events emits incorrect information

Description: When the `liquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMindAndTran` function, values are logged in an incorrect other. The `poolTokensToDeposit` parameter should go in the third parameter position instead of the `wethToDeposit` parameter which should go second.

Impact: Because the event is emitted incorrectly, this can lead to a malfunctioning of off-chain functions.

Recommended Mitigation

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

[L-2] `public` functions not used internally could be marked `external`

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally.

1 Found Instances

- Found in src/TSwapPool.sol Line: 301

  ```
      function swapExactInput(
  ```

[L-3] `TSwapPool::swapExactInput` will result in an incorrect return value being passed on to the user.

Description: The function is meant to return the calculated amount of tokens to be received by a caller based on a given input amount. However, the output variable is not assigned a value in the function as retrieved from the `getOutputAmountBasedOnInput` function that is called.

Impact: The return value will always be zero which is incorrect information being passed on to the user.

Proof of Concept: Please include the following test in `TSwapPool.t.sol` to illustrate the issue

Code

```
function testSwapExactInputReturnsZero() public liquidityProvided {
        vm.startPrank(user);
        uint256 inputAmount = 2e18;
        uint256 expectedOutputAmount = 0;
        uint256 minOutputAmount = 1e18;
        poolToken.approve(address(pool), type(uint256).max);
        // Let the user swap
        uint256 outputAmount = pool.swapExactInput(
            poolToken,
            inputAmount,
            weth,
            minOutputAmount,
            uint64(block.timestamp)
        );

```

```
        assertEq(outputAmount, expectedOutputAmount, "Output amount
            should be zero.");

    }
```

Recommended Mitigation

Code

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

-        uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
    inputReserves, outputReserves);
+         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__PoolDoesNotExist` is not used and should preferably be removed.

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

[I-2] `PoolFactory::constructor(address wethToken)` does not have a zero check for the wethToken address.

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

```
    }
```

[I-3] `PoolFactory::createPool()` uses `IERC20(tokenAddress).name()` instead of `.symbol()`

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

[I-4] `TSwapPool::Swap()` event has more than three parameters but only one event is indexed. See below for details and other instances.

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

4 Found Instances

- Found in src/PoolFactory.sol Line: 35

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

- Found in src/TSwapPool.sol Line: 53

  ```
      event LiquidityAdded(
  ```

- Found in src/TSwapPool.sol Line: 58

  ```
      event LiquidityRemoved(
  ```

- Found in src/TSwapPool.sol Line: 63

  ```
      event Swap(
  ```

[I-5] `TSwapPool::constructor` is lacking a zero address check for the `poolToken` and `wethToken` variables.

```
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-6] `TSwapPool::deposit` does not use the `poolTokenReserves` variable. Consider removing it.

```
if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this))
                ;
            //@audit-gas: This line is not used, consider removing it.
-            uint256 poolTokenReserves = i_poolToken.balanceOf(address(
   this));
```