Protocol Audit Report

Version 1.0

Sachet Dhanuka

July 7, 2024

Protocol Audit Report

Sachet Dhanuka

July 7, 2023

Prepared by: SachetDhanuka Lead Auditors: - Patrick Collins - Sachet Dhanuka

Table of Contents

- Medium
  * [M-1] `TSwapPool::deposit` is lacking `deadline` check causing transactions to execute even after the deadline
  * [M-2] WeirdERC20 `FeeOnTransfer` tokens break the protocol invariant `x * y = k`
- Low
  * [L-1] `TSwapPool::LiquidityAdded` event emitted in `TSwapPool::_addLiquidityMintAndTransfer` private function is out of order
  * [L-2] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given.
- Informational
  * [I-1] `PoolFactory::PoolFactory_PoolDoesNotExist` error is not used and should be removed
  * [I-2] Lacking a zero address check of input `PoolFactory::wethToken` in `constructor`
  * [I-3] `PoolFactory::liquidityTokenSymbol` should use `.symbol()` instead of `.name()`
  * [I-4] Events have missing `indexed` fields

Protocol Summary

TSwapPool is a decentralized AMM of WETH/POOLTOKENS. Each TSwapPool is a market for a pair of tokens of which one is WETH. These decentralized exhanges can be created using PoolFactory.sol.This project is meant to be a permissionless way for users to swap assets between each other at a fair price.The invaraiant of the protocol is `x * y = k` which should not change during swap. Each swap charges a 0.003% percent of fees by the users which is kind of an incentive for the liquidity providers of the pool.

Disclaimer

Sachet Dhanuka and 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
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

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

Issues found

Severtity	Number of issues found
High	4
Medium	2
Low	2
Info	4
Total	12

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from the user, resulting in a lot of fees

Description The `TSwapPool::getInputAmountBasedOnOutput` is intended to calculate the amount of tokens a user should deposit, given the amount of output tokens he wants as an output. However, the function miscalculates the resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1000.

Impact Protocol takes more fees than expected from its users.

Proof Of Concept 1. Make the liquidity provider supply the Pool with initial liquidity, 100 WETH 100 POOLTOKENS. 2. Prank a swapper to swap 1 WETH by calling `swapExactOutput` function.

```
    uint256 actualinputAmount = pool.getInputAmountBasedOnOutput(1e18,
        startingPoolToken,startingWeth);
    // 10131404313951956688
    uint256 expectedinputAmount =  ((startingPoolToken * 1e18) * 1000) /
            ((startingWeth - 1e18) * 997);
    // 1013140431395195688O
```

POC

Place the following test in `TSwapPoolTest.t.sol`:

```
        function testFlawedSwapExactOutput()public{
        uint256 startingWeth = 100e18;
        uint256 startingPoolToken = 100e18;
        uint256 initialPoolTokenOfSwapper = 15e18;
        // Initial liquidity into the pool
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), startingWeth);
        poolToken.approve(address(pool), startingPoolToken);
        pool.deposit(startingWeth, startingWeth, startingPoolToken,
            uint64(block.timestamp));
        vm.stopPrank();
        // A swapper who wants to extract 1 WETH as Output
        address swapper = makeAddr("swapper");
        poolToken.mint(swapper,initialPoolTokenOfSwapper);
        vm.startPrank(swapper);
        poolToken.approve(address(pool),type(uint256).max);
        // Actual Input Amount taken for 1 WETH as Output
        uint256 actualinputAmount = pool.getInputAmountBasedOnOutput(1
            e18,startingPoolToken,startingWeth);
        // Expected Input Amount
        uint256 expectedinputAmount =  ((startingPoolToken * 1e18) *
            1000) /
             ((startingWeth - 1e18) * 997);
        pool.swapExactOutput(poolToken,weth,1e18,uint64(block.timestamp
            ));
        console.log("Expected Input Amount : ",expectedinputAmount);
        console.log("Actual Input Amount...: ",actualinputAmount);
        console.log("Balance PoolTokens of swapper : ",poolToken.
            balanceOf(swapper));

        assertEq(poolToken.balanceOf(address(pool))-100e18,
            actualinputAmount);
        // The swapper had to pay 10 times of what he was supposed to
        // the swap which should had cost him ~1 ETH costed him ~ 10
            ETH!
    }
```

Recommended Mitigation Use `1_000` instead of `10_000` while calaculating `inputAmount` in `TSwapPool::getInputBasedOnOutput`

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

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` can potentially cause the users to pay way more tokens than they desire to pay

Description The `swapExactOutput` function does not ask for the maximum threshold amount the user is willing to pay for the exact number of tokens they want to buy. So there is a possibility that the demand of the buying tokens might experiance a sudden surge from the time of putting in the order till the time of its execution. So if the slippage protection is not set, the user might end up paying way more than he desired to.

The way `TSwapPool::swapExactInput` has `minOutputAmount` specified, similarly the `swapExactOutput` should specify a `maxInputAmount` for slippage protection.

Impact If the market conditions change before the transaction processes, the user might end up paying way more than he was willing to pay.

Proof Of Concept

1. Initial Liquidity is provided into the pool, suppose 50 WETH and 100 POOLTOKENS.
2. A user places a transaction by calling `swapExactOutput` to extract 5 WETH from the pool.
3. The transaction stays in the memepool for a while and during this time a market operator manipulates the price.
4. the market operator extracts 25 WETH from the pool and incentivises the miner to include the transaction in the block.
5. So when the transaction of the user is executed, the dynamics of the pool has changed completely.

```
    Expected Input Amount :   11144544745347152568   ~11.1ETH
    Actual Input Amount :     50225903387192671293   ~50.2ETH
```

6. The user ends up paying 5 times more than he desired to pay while he made the transaction.

POC

Place the following code in `TSwapPool.t.sol`:

```
function testSlippageProtectionInswapExactOutput()public{
        uint256 startingWeth = 50e18;
        uint256 startingPoolToken = 100e18;
        address marketOperator = makeAddr("MO");
        poolToken.mint(marketOperator,200e18);
        weth.mint(marketOperator,200e18);

        // Initial liquidity into the pool
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), startingWeth);
        poolToken.approve(address(pool), startingPoolToken);
        pool.deposit(startingWeth, startingWeth, startingPoolToken,
            uint64(block.timestamp));
        vm.stopPrank();

        // A user places a transaction at this point to withdraw 5 WETH
             as output from the pool
        uint256 expectedInputAmount = pool.getInputAmountBasedOnOutput
            (5e18,poolToken.balanceOf(address(pool)),weth.balanceOf(
            address(pool)));
        // Considering that the bug in this function was removed and 10
            _000 was replaced with 1_000
        // But before the transaction is executed a massive trade is
            executed which changes the market!
        vm.startPrank(marketOperator);
        // Withdraws 25 WETH from the pool
        poolToken.approve(address(pool),type(uint256).max);
        pool.swapExactOutput(poolToken,weth,25e18,uint64(block.
            timestamp));
        vm.stopPrank();

        uint256 actualInputAmount = pool.getInputAmountBasedOnOutput(5
            e18,poolToken.balanceOf(address(pool)),weth.balanceOf(
            address(pool)));
        uint256 userpoolTokenBalance = poolToken.balanceOf(user);
        // The transaction user placed is executed now!
        vm.startPrank(user);
        poolToken.approve(address(pool),type(uint256).max);
        pool.swapExactOutput(poolToken,weth,5e18,uint64(block.timestamp
            ));
        vm.stopPrank();

        assertEq(poolToken.balanceOf(user),userpoolTokenBalance-
            actualInputAmount);
        console.log("Expected Input Amount : ",expectedInputAmount);
        console.log("Actual Input Amount : ",actualInputAmount);
    }
```

Recommended Mitigation We should include a `maxInputAount` so that the user only has to spend upto that limit amount, and can predict on how much they will to spend.

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

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

Description The `sellPoolTokens` function is intended to allow users to sell pool tokens and receive weth in exchange. The 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 it calling the `swapExactOutput` function instead of `swapExactInput` function. Here the users are specifying the exact amount of input they want to sell to the pool in exchange of certain amount of output.

Inpact Users will swap the wrong amount of tokens, which is a severe disruption of protocol's functionality.

Proof Of Concept

User ends up paying more pool tokens than he wanted to :

```
    Expected input pool tokens :  10000000000000000000 = 10 ETH
    Actual input pool tokens :    25075225677031093279 ~ 25 ETH
```

POC

Place the following code in `TSwapPool.t.sol`:

```
function testsellPoolTokens()public{
        uint256 startingPoolToken = 100e18;
        uint256 startingWeth = 50e18;

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool),type(uint256).max);
        poolToken.approve(address(pool),type(uint256).max);
        pool.deposit(startingWeth,startingWeth,startingPoolToken,uint64
            (block.timestamp));
        vm.stopPrank();
        // User wants to input 10e18 POOL TOKENS
        uint256 expectedInputPoolTokens = 10e18;
        uint256 userStartingPoolTokenBalance = poolToken.balanceOf(user
            );
        uint256 actualInputPoolTokens = pool.
            getInputAmountBasedOnOutput(expectedInputPoolTokens,
            poolToken.balanceOf(address(pool)),weth.balanceOf(address(
            pool)));
        vm.startPrank(user);
        poolToken.approve(address(pool),type(uint256).max);
        pool.sellPoolTokens(expectedInputPoolTokens);
        vm.stopPrank();
        // This function calls `swapExactoutput`! So instead of user
            selling 10 pooltokens, they end up asking for 10 weth tokens
            .
        uint256 userEndingPoolTokenBalance = poolToken.balanceOf(user);

        assertEq(userEndingPoolTokenBalance,
            userStartingPoolTokenBalance-actualInputPoolTokens);
        console.log("Expected input pool tokens : ",
            expectedInputPoolTokens);
        console.log("Actual input pool tokens : ",actualInputPoolTokens
            );
    }
```

Recommended Mitigation Consider changing the implementation of using `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept new parameter `minWethToOutput`, which will be passed in `swapExactInput` .

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

Additionally, it would be wise to add a deadline to the current function as there is no deadline.

[H-4] 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 product 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 follow 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 a lot of 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 untill 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, we should account for the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we 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

[M-1] `TSwapPool::deposit` is lacking `deadline` check causing transactions to execute even after the deadline

Description The `deposit` function accepts a `deadline` parameter which according to the documentation states "The deadline for the transaction to be completed by". However, this parameter is never used. This causes the operations that add liquidity to the pool, to be executed at unexpected times, in market conditions where the deposit rate is not favourable.

Impact Transactions can be executed at unfavourable market conditions, despite adding the `deadline`.

Proof Of Concept

Recommended Mitigation Consider making the following change to the function

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

[M-2] WeirdERC20 `FeeOnTransfer` tokens break the protocol invariant `x * y = k`

Description Suppose a user calls the `TSwapPool::swapExactInput` function to exchange weth for some of its poolTokens. The pool calculates the weth to send based on the expected input poolTokens. It then using `safetransfer` function in the `TSwapPool::_swap`, transfers the exact amount of weth based on the calculation.

The problem occurs during the transfer of poolTokens from user to TSwapPool. This weirdERC20 token charges transaction fee, burning some share of the tokens before the amount is credited in TSwapPool address. The TSwapPool does not receive the expected amount of poolTokens as per the amount of weth it has given out, breaking the invariant of the protocol.

Impact The amount of poolTokens the TSwapPool receives during an exhange for weth is less than what it was expected to receive, thereby breaking the invariant of the protocol.

Proof Of Concept Deploy a WeirdERC20 token and create a TSwapPool of WETH and WEIRDERC20. This WeirdERC20 is a deflationary token. It charges a 10 percent fee on every 5th transaction that occurs. The fee tokens are burnt and the rest is send as per the normal exection process.

```
Expected DeltaX :  100000000000000000000 // User sent out 100 ETH worth of WeirdERC20
Actual DeltaX   :  90000000000000000000  // TSwapPool receives 90 ETH worth of WeirdERC20
```

POC

Place the following code in `TSwapPool.t.sol`:

```
    function test_FeeOnTransferIssue()public{
        TSwapPool tswapPool;
        int256 initialX = 100e18;
        int256 initialY = 50e18;

        vm.startPrank(liquidityProvider);
        WeirdERC20 weirdERC20 = new WeirdERC20(); // 1st Transaction
        tswapPool = new TSwapPool(address(weirdERC20), address(weth), "LTokenA", "LA");
        weirdERC20.approve(address(tswapPool),type(uint256).max);
        weth.approve(address(tswapPool),type(uint256).max);
        tswapPool.deposit(uint256(initialY),0,uint256(initialX),uint64(block.timestamp)); // 2nd Transaction
        vm.stopPrank();

        weirdERC20.mint(user,200e18); // 3rd Transaction

        vm.startPrank(user);
        weirdERC20.approve(address(tswapPool),type(uint256).max);
        tswapPool.swapExactInput(weirdERC20,5e18,weth,0,uint64(block.timestamp)); // 4th Transaction
        vm.stopPrank();

        int256 startingX = int256(weirdERC20.balanceOf(address(tswapPool)));
        int256 startingY = int256(weth.balanceOf(address(tswapPool)));

        int256 expectedDeltaX = 100e18;
        uint256 output =  tswapPool.getOutputAmountBasedOnInput(uint256(expectedDeltaX),uint256(startingX),uint256(startingY));
        int256 expectedDeltaY = (-1) * int256(output);

        // 5th Transaction
        vm.startPrank(user);
        tswapPool.swapExactInput(weirdERC20,uint256(expectedDeltaX),weth,output,uint64(block.timestamp));
        vm.stopPrank();

        int256 endingX = int256(weirdERC20.balanceOf(address(tswapPool)));
        int256 endingY = int256(weth.balanceOf(address(tswapPool)));

        int256 actualDeltaX = endingX - startingX;
        int256 actualDeltaY = endingY - startingY;

        console.log("Expected DeltaX : ",uint256(expectedDeltaX));
        console.log("Actual DeltaX : ",uint256(actualDeltaX));

        assertEq(expectedDeltaX,actualDeltaX);
        assertEq(expectedDeltaY,actualDeltaY);
    }
```

Recommended Mitigation During the exchange and transfer in the `TSwapPool::_swap` function, an internal check can be performed, calculating the before and after balance of the TSwapPool address in the `inputToken`. This prevents the contract from accepting less than what was expected, by reverting the transaction.

```
.
.
.
 emit Swap(
            msg.sender,
            inputToken,
            inputAmount,
            outputToken,
            outputAmount
        );
+       uint256 beforeBalance = inputToken.balanceOf(address(pool));
        inputToken.safeTransferFrom(msg.sender, address(this),
            inputAmount);
+        uint256 afterBalance = inputToken.balanceOf(address(pool));
+        if((afterBalance-beforeBalance) < inputAmount){
+            revert();
+        }
         outputToken.safeTransfer(msg.sender, outputAmount);
     }
```

Low

[L-1] `TSwapPool::LiquidityAdded` event emitted in `TSwapPool::_addLiquidityMintAndTransfer` private function is out of order

Description When the `LiquidityAdded` event is emitted in `TSwapPool::_addLiquidityMintAndTransfe` function, it logs values in incorrect order. The `poolTokensToDeposit` value should go in the third parameter position, while the `wethToDeposit` should go in the second.

```
function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
       )private {
        _mint(msg.sender, liquidityTokensToMint);
@>      emit LiquidityAdded(msg.sender, poolTokensToDeposit,
    wethToDeposit);
        i_wethToken.safeTransferFrom(msg.sender, address(this),
            wethToDeposit);
        i_poolToken.safeTransferFrom(
            msg.sender,
            address(this),
            poolTokensToDeposit
        );
    }
```

Impact Event emission is incorrect, leading to off-chain function potentially malfunctioning

Recommended Mitigation

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

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

Description The `swapExactInput` 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, nor uses an explicit return statement.

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

Proof Of Concept

POC

Place the following code in `TSwapPool.t.sol`:

```
function testswapExactInputReturnsZero()public{
        uint256 startingWeth = 100e18;
        uint256 startingPoolToken = 100e18;
        address swapper = makeAddr("swapper");
        weth.mint(swapper,10e18);

        // Initial liquidity into the pool
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), startingWeth);
        poolToken.approve(address(pool), startingPoolToken);
        pool.deposit(startingWeth, startingWeth, startingPoolToken,
            uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(swapper);
        weth.approve(address(pool),10e18);
        uint256 outputPoolToken = pool.getOutputAmountBasedOnInput(1e18
            ,weth.balanceOf(address(pool)),poolToken.balanceOf(address(
            pool)));
        uint256 output = pool.swapExactInput(weth,1e18,poolToken,
            outputPoolToken,uint64(block.timestamp));
        vm.stopPrank();
        assertEq(output,0);

    }
```

Recommended Mitigation

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
```

```
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
-       returns (uint256 output)
+       returns (uint256 outputAmount)
    {...
```

Informational

[I-1] `PoolFactory::PoolFactory_PoolDoesNotExist` error is not used and should be removed

```
    contract PoolFactory {
    error PoolFactory__PoolAlreadyExists(address tokenAddress);
-   error PoolFactory__PoolDoesNotExist(address tokenAddress);
```

[I-2] Lacking a zero address check of input `PoolFactory::wethToken` in `constructor`

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

[I-3] `PoolFactory::liquidityTokenSymbol` should use `.symbol()` instead of `.name()`

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

[I-4] Events have missing `indexed` fields

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: 36

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

- Found in src/TSwapPool.sol Line: 52

  ```
      event LiquidityAdded(
  ```

- Found in src/TSwapPool.sol Line: 57

  ```
      event LiquidityRemoved(
  ```

- Found in src/TSwapPool.sol Line: 62

  ```
      event Swap(
  ```