Protocol Audit Report

Jordan J. Solomon

July 04, 2024

Ways of the Learner

JJS Learnings

T-Swap Audit Report

Version 1.0

Jordan J. Solomon

July 4, 2024

Protocol Audit Report

Jordan J. Solomon

July 04, 2024

Prepared by: Jordan J. Solomon

Table of Contents

- Informational
  - [I-1] PoolFactory::wethAddress, TSwapPool::wethToken, TSwapPool::poolToken lacks address(0) check, can cause lose of ownership
  - [I-2] In TSwapPool::swapExactInput and TSwapPool::totalLiquidityTokenSupply, public functions not used internally could be marked external
  - [I-3] Define and use constant variables instead of using literals
  - [I-4] Large literal values multiples of 10000 can be replaced with scientific notation
  - [I-5] Unused Custom Error PoolFactory::PoolFactory___PoolDoesNotExist
  - [I-6] Using .name() instead of .symbol() in PoolFactory::createPool, causing an unwanted symbol
  - [I-7] No reason to emit TSwapPool.sol::MINIMUM_WETH_LIQUIDITY in TSwapPool::TSwapPool___WethDepositAmountTooLow event
  - [I-8] Changing a variable after an external call in TSwapPool::deposit
  - [I-9] Missing NatSpec in TSwapPool::getOutputAmountBasedOnInput and TSwapPool::getInputAmountBasedOnOutput function
  - [I-10] TSwapPool::deadline param in TSwapPool::swapExactOutput is missing in NatSpec
  - [I-11] Events are missing indexed fields
  - [I-12] Unused variables in TSwapPool.sol

Protocol Summary

This project is meant to be a permission less 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. It is similar to Uniswap. To understand Uniswap, please watch this video: Uniswap Explained

Disclaimer

The JJS 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

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda

```
./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

In this audit I have learned more about using fuzz/invariant testing. This helps automate the audit process by finding bugs without actually looking at the code. I have done more manual and static review.

Issues found

Severity	Amount
High	4
Medium	2
Low	2
Info	11
Total	19

Findings

High

[H-1] `TSwapPool::_swap` incentive breaks the invariant of `x * y = k`

Description: In the TSwapPool::_swap there is an incentive to keep the user, using "T-Swap". That incentive gifts the user an extra token after 10 swaps. However, we have an invariant where:

- x: The balance of the pool token
- y: The balance of WETH
- k: The constant product of the two balances

The ratio between the tokens need to remain the same even after swaps, deposits and withdraws, hence the k constant. However, this incentive breaks this invariant by affecting the ration with an extra token per user after making 10 swaps.

Here is the code that 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.

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.

Place this test in TSwaPool.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 incentive after 10 swaps.

```
- 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);
- }
```

[H-2] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees

Description: In TSwapPool::getInputAmountBasedOnOutput we calculate the input based on the output that the user passes. However, we use magic numbers. Instead of multiplying by a 1_000 we multiply by 10_000!

Impact: This is causing an enormous amount of fees on the user. In addition, miss use of best practice is causing this issue!

Proof of Concept:

Here is a test to see this in action:

Proof of Code

```
function testFlawedSwapExactOutput() public {
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        pool.deposit({
            wethToDeposit: initialLiquidity,
            minimumLiquidityTokensToMint: 0,
            maximumPoolTokensToDeposit: initialLiquidity,
            deadline: uint64(block.timestamp)
        });
        vm.stopPrank();

        // User has 11 pool tokens
        address someUser = makeAddr("someUser");
        uint256 userInitialPoolTokenBalance = 11e18;
        poolToken.mint(someUser, userInitialPoolTokenBalance);
        vm.startPrank(someUser);

        // Users buys 1 WETH from the pool, paying with pool tokens
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, 1 ether, uint64(block.timestamp));

        // Initial liquidity was 1:1, so user should have paid ~1 pool token
        // However, it spent much more than that. The user started with 11 token
        assertLt(poolToken.balanceOf(someUser), 1 ether);
        vm.stopPrank();

        // The liquidity provider can rug all funds from the pool now,
        // including those deposited by user.
        vm.startPrank(liquidityProvider);
        pool.withdraw(
            pool.balanceOf(liquidityProvider),
            1, // minWethToWithdraw
            1, // minPoolTokensToWithdraw
            uint64(block.timestamp)
        );

        assertEq(weth.balanceOf(address(pool)), 0);
        assertEq(poolToken.balanceOf(address(pool)), 0);
    }
```

Recommended Mitigation: Use constants for magic numbers. Helps read the code better, and reduces these mistakes! In addition, if we ever need to make changes to these number, we only need to do that in one place rather than everywhere it is used.

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-         return ((inputReserves * outputAmount) * 10000) / ((outputReserves - ou
+         return ((inputReserves * outputAmount) * FEE_PRECISION) / ((outputReser
    }
```

[H-3] No slippage protection in `TSwapPool::swapExactOutput`, causing potential lower token to be received by user

Description: The swapExactOutput function does not include any sort of 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 is 1,000 USDC
2. User inputs a swapExactOutput looking for 1 WETH

- inputToken => USDC
- outputToken => WETH
- outputAmount => 1
- deadline = block.timestamp

3. The function does not offer a maxInput amount
4. As the transaction is pending in the mempool, the market changes! And the price moves HUGE -> 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: Add a check for maxInputAmount:

Possible Fix

```
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, o

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

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

[H-4] `TSwapPool::sellPoolToken` 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're 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 is the one that should be called. Because 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 protcol functionality.

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 (ie minWethToReceive to be passed to swapExactInput).

```
      function sellPoolTokens(
          uint256 poolTokenAmount,
+         uint256 minWethToReceive,
          ) external returns (uint256 wethAmount) {
-           return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount, uint6
+           return swapExactInput(i_poolToken, poolTokenAmount, i_wethToken, minWet
    }
```

Medium

[M-1] No use of the `TSwapPool::deadline` variable and the `TSwapPool::revertIfDeadlinePassed` modifier in `TSwapPool::deposit`, causing transaction to complete even after the deadline.

Description: The deposit function accepts a deadline parameter, which according to the documentation is "The deadline for the transaction to be completed by". However, this parameter is never used. As a consequence, operations that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavorable.

Impact: Transaction could be sent when market conditions are unfavourable to deposit, even when adding a deadline parameter.

Proof of Concept: the TSwapPool::deadline is never used.

Recommended Mitigation: Consider making the following change to the function.

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

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

Description: Weird ERC20 and ERC777 tokens can break protocol invariant by changing the ratio of $x * y = k$.

Impact: Ruining the protocol functionality by using these tokens.

Proof of Concept:

Recommended Mitigation:

Low

[L-1] Wrong order of variables `TSwapPool::poolTokensToDeposit` and `TSwapPool::wethToDeposit` when emitting `TSwapPool::LiquidityAdded` event

Description: When defining the TSwapPool::LiquidityAdded event it is set to have TSwapPool::wethWithdrawn before TSwapPool::poolTokensWithdrawn. However, in TSwapPool::_addLiquidityMintAndTransfer the event is emitted in the wrong order.

Impact: Event emission is incorrect, leading to off-chain functions and potentially malfunctioning.

Recommended Mitigation:

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

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

Description: The swapExactInput function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return value output it is never assigned a value, nor uses an explicit return statement.

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

Proof of Concept: Here is a the breakdown of the proof and the test afterwards to implement in TSwapPoolTest.t.sol:

1. The liquidityProvider deposits tokens in a new pool.
2. Minting 1 poolToken for a new user.
3. Performing the swap using swapExactInput function.
4. Checking if the returned variable is equal to 0

Proof of Code

```
function testSwapExactInputAlwaysReturnsZero() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), type(uint256).max);
        poolToken.approve(address(pool), type(uint256).max);
        pool.deposit(STARTING_Y, STARTING_Y, STARTING_Y, uint64(block.timestamp)
        vm.stopPrank();

        uint256 newAmount = 1e18;
        poolToken.mint(user, newAmount);
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        uint256 returned = pool.swapExactInput(poolToken, newAmount, weth, 0, ui
        vm.stopPrank();

        assertEq(returned, 0);
    }
```

Recommended Mitigation:

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

-         uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount, inputRe
+         output = getOutputAmountBasedOnInput(inputAmount, inputReserves, output

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

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

Informational

[I-1] `PoolFactory::wethAddress`, `TSwapPool::wethToken`, `TSwapPool::poolToken` lacks `address(0)` check, can cause lose of ownership

Description: The address passed through the constructor could be an address 0.

Impact: Losing ownership of the contract.

Proof of Concept:

3 Found Instances

- Found in src/PoolFactory.sol Line: 43

  ```
  i_wethToken = wethToken;
  ```

- Found in src/TSwapPool.sol Line: 81

  ```
  i_wethToken = IERC20(wethToken);
  ```

- Found in src/TSwapPool.sol Line: 82

```
        i_poolToken = IERC20(poolToken);
```

[I-2] In `TSwapPool::swapExactInput` and `TSwapPool::totalLiquidityTokenSupply`, `public` functions not used internally could be marked `external`

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

Impact: - Gas efficiency - When function parameters are large arrays or complex types, external functions can access calldata more directly. This is because in Ethereum, the calldata is stored in a separate area and does not need to be copied to memory when accessed directly. For public functions, even if they are called externally, the parameters have to be copied from calldata to memory, consuming more gas. Therefore, for functions that accept large or complex data types and are called externally, marking them as external reduces gas costs. - Clear Intent and Security: By using external, it's clear that the function is designed to be accessed only from outside the contract. This can help prevent errors or misuse in contract development by signaling to other developers that the function should not be attempted to be called internally. This can also prevent security risks associated with unexpected internal calls. - Interface Clarity: Marking functions as external explicitly defines the intended interaction interface with other contracts and external accounts. This makes the smart contract easier to understand and integrate within the broader dApp ecosystem.

Possible Mitigation: Change it to external

[I-3] Define and use `constant` variables instead of using literals

Description: If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

Impact:

- Compile-time Constants: Declaring a value as a constant state variable allows the Solidity compiler to replace references to the variable with the constant value directly in the bytecode. This can lead to reduced gas costs because accessing a constant requires less gas than reading a non-constant state variable.
- Reduced Deployment Costs: Since constant values do not take up storage space, they do not contribute to the contract's deployment costs, unlike regular state variables that do require storage.
- Easy Updates: Using a constant for repeated values means that you only need to update the value in one place if it ever needs to change. This simplifies maintenance and reduces the risk of errors that can occur if the value is updated inconsistently across the contract.
- Improved Readability: Constants are usually given descriptive names that convey their purpose, which can make the code more readable and

understandable. This is particularly useful for others reviewing or auditing the code.

Proof of Concept:

4 Found Instances

- Found in src/TSwapPool.sol Line: 231

```
        uint256 inputAmountMinusFee = inputAmount * 997;
```

- Found in src/TSwapPool.sol Line: 248

```
        return ((inputReserves * outputAmount) * 10000) / ((outputReserves -
```

[I-4] Large literal values multiples of 10000 can be replaced with scientific notation

Description: Use e notation, for example: 1e18, instead of its full numeric value.

Impact: - Readability: Representing large numbers in scientific notation can improve the readability and maintainability of code. It makes it clear at a glance the scale of the number, especially when dealing with values like constants used in financial calculations. - Error Reduction: Using scientific notation can help reduce errors in typing long digit numbers, as it's easier to mistype or miscount zeros in large numbers.

Proof of Concept:

3 Found Instances

- Found in src/TSwapPool.sol Line: 36

```
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
```

- Found in src/TSwapPool.sol Line: 248

```
        return ((inputReserves * outputAmount) * 10000) / ((outputReserves -
```

- Found in src/TSwapPool.sol Line: 331

```
            outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
```

[I-5] Unused Custom Error `PoolFactory::PoolFactory__PoolDoesNotExist`

Description: It is recommended that the definition be removed when custom error is unused.

Impact: - Increased Deployment Cost: Every line of code, including error definitions, contributes to the overall bytecode size of a smart contract. Deploying more bytecode means higher gas costs. If the error is not used, you are essentially paying for something that doesn't serve any function in the contract. - Code Clutter: Unused code, including errors, can make the contract less readable and harder to maintain. It can lead to confusion about the contract's functionality and make audits more challenging, as auditors need to verify whether each part of the code is used and secure. - Potential for Bugs: While unused code generally doesn't introduce direct bugs, it can lead to misconceptions about how the contract is supposed to behave. Developers or future maintainers might assume that certain validations or checks are performed based on the presence of these errors, when in fact they are not. - Best Practices: It's a good coding practice to remove any unused or redundant code to keep the contract clean and efficient. This not only helps in reducing the cost but also aids in the readability and auditability of the contract.

Proof of Concept:

1 Found Instances

- Found in src/PoolFactory.sol Line: 22

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

[I-6] Using `.name()` instead of `.symbol()` in `PoolFactory::createPool`, causing an unwanted symbol

Description: In the PoolFactory::createPool function, where we concatenate strings, we add the .name() to the PoolFactory::liquidityTokenSymbol instead of the .symbol().

Impact: This means that the symbol will be wrong.

Proof of Concept:

1 Found Instances

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

[I-7] No reason to emit `TSwapPool.sol::MINIMUM_WETH_LIQUIDITY` in `TSwapPool::TSwapPool__WethDepositAmountTooLow` event

Description: Emitting a constant is redundant as it does not change. If needed, anyone can look at the contract wants it's deployed and see the value.

Impact: Redundancy in code, simplify the code and best practice.

Recommended Mitigation:

Changes can be made

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

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

[I-8] Changing a variable after an external call in `TSwapPool::deposit`

Description: To prevent attacks, such as Reentrancy attacks, it is best practice to make any changes to our code before any external calls. However, this change is not for a state variable, which makes it am Informational Finding and not higher.

Impact: Not best practice.

Recommended Mitigation:

Possible Change

```
} else {
-       _addLiquidityMintAndTransfer(wethToDeposit, maximumPoolTokensToDeposit, wet
-       liquidityTokensToMint = wethToDeposit;

+       liquidityTokensToMint = wethToDeposit;
+       _addLiquidityMintAndTransfer(wethToDeposit, maximumPoolTokensToDeposit, wet

            }
```

[I-9] Missing NatSpec in `TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool::getInputAmountBasedOnOutput` function

Description: Need to give some documentation, especially for important functions.

Impact: Less understanding of what the function is supposed to do.

[I-10] `TSwapPool::deadline` param in `TSwapPool::swapExactOutput` is missing in NatSpec

Description: We need to explain all of the parameters in the NatSpec

[I-11] Events are missing `indexed` fields

Description: 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.

Impact:

- Filtering Events: The primary role of the indexed keyword in Solidity event parameters is to enable efficient filtering. Up to three parameters can be indexed per event. These indexed parameters are then searchable in the blockchain's logs, facilitating the retrieval of events based on indexed attributes.
- Without indexed: If event parameters are not indexed, users and applications would need to download and parse all instances of that event to search for specific records, which can be significantly slower and more costly in terms of computing resources.
- Event Storage: While the indexed fields themselves do not directly impact the gas cost associated with emitting an event, the overall structure and content of events can influence transaction costs. Efficient indexing helps in reducing the need to emit excess data if only key parameters need to be quickly accessible.
- Access Patterns: Developers use indexed parameters to create a more accessible log structure that applications can query against efficiently. Without indexing, developers might face limitations in how effectively they can create user interfaces or backend services that react to contract activities.

Proof of Concept:

4 Found Instances

- Found in src/PoolFactory.sol Line: 35

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

- Found in src/TSwapPool.sol Line: 43

  ```
  event LiquidityAdded(address indexed liquidityProvider, uint256 wethDepo
  ```

- Found in src/TSwapPool.sol Line: 44

  ```
  event LiquidityRemoved(address indexed liquidityProvider, uint256 wethWi
  ```

- Found in src/TSwapPool.sol Line: 45

  ```
  event Swap(address indexed swapper, IERC20 tokenIn, uint256 amountTokenI
  ```

[I-12] Unused variables in `TSwapPool.sol`

Description: Unused variables in the contract should be removed from the codebase.

Impact: They make our code less readable, and make our contract less gas efficient.

Recommended Mitigation:

# Found Instances

- Found in src/TSwapPool.sol Line: 114

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

- Found in src/TSwapPool.sol Line: 284

  ```
  uint256 output
  ```