T-Swap Audit Report

Prepared by: Jack Landon

Prepared by: Jack Landon on 2024-08-17 Lead Auditor:

- Jack Landon

Table of Contents

Protocol Summary

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

TSwap Pools

The protocol starts as simply a `PoolFactory` contract. This contract is used to create new "pools" of tokens. It helps make sure every pool token uses the correct logic. But all the magic is in each `TSwapPool` contract.

You can think of each `TSwapPool` contract as it's own exchange between exactly 2 assets. Any ERC20 and the WETH token. These pools allow users to permissionlessly swap between an ERC20 that has a pool and WETH. Once enough pools are created, users can easily "hop" between supported ERC20s.

For example:

1. User A has 10 USDC
2. They want to use it to buy DAI
3. They `swap` their 10 USDC -> WETH in the USDC/WETH pool
4. Then they `swap` their WETH -> DAI in the DAI/WETH pool

Every pool is a pair of `TOKEN X` & `WETH`.

There are 2 functions users can call to swap tokens in the pool.

- `swapExactInput`
- `swapExactOutput`

We will talk about what those do in a little.

Liquidity Providers

In order for the system to work, users have to provide liquidity, aka, "add tokens into the pool".

Why would I want to add tokens to the pool?

The TSwap protocol accrues fees from users who make swaps. Every swap has a `0.3` fee, represented in `getInputAmountBasedOnOutput` and `getOutputAmountBasedOnInput`. Each applies a `997` out of `1000` multiplier. That fee stays in the protocol.

When you deposit tokens into the protocol, you are rewarded with an LP token. You'll notice `TSwapPool` inherits the `ERC20` contract. This is because the `TSwapPool` gives out an ERC20 when Liquidity Providers (LP)s deposit tokens. This represents their share of the pool, how much they put in. When users swap funds, 0.03% of the swap stays in the pool, netting LPs a small profit.

LP Example

1. LP A adds 1,000 WETH & 1,000 USDC to the USDC/WETH pool
   1. They gain 1,000 LP tokens
2. LP B adds 500 WETH & 500 USDC to the USDC/WETH pool
   1. They gain 500 LP tokens
3. There are now 1,500 WETH & 1,500 USDC in the pool
4. User A swaps 100 USDC -> 100 WETH.
   1. The pool takes 0.3%, aka 0.3 USDC.
   2. The pool balance is now 1,400.3 WETH & 1,600 USDC
   3. aka: They send the pool 100 USDC, and the pool sends them 99.7 WETH

Note, in practice, the pool would have slightly different values than 1,400.3 WETH & 1,600 USDC due to the math below.

Core Invariant

Our system works because the ratio of Token A & WETH will always stay the same. Well, for the most part. Since we add fees, our invariant technially increases.

`x * y = k`

- x = Token Balance X
- y = Token Balance Y
- k = The constant ratio between X & Y

```
y = Token Balance Y
x = Token Balance X
x * y = k
x * y = (x + ∆x) * (y − ∆y)
∆x = Change of token balance X
∆y = Change of token balance Y
β = (∆y / y)
α = (∆x / x)

Final invariant equation without fees:
∆x = (β/(1-β)) * x
∆y = (α/(1+α)) * y

Invariant with fees
ρ = fee (between 0 & 1, aka a percentage)
γ = (1 - p) (pronounced gamma)
∆x = (β/(1-β)) * (1/γ) * x
∆y = (αγ/1+αγ) * y
```

Our protocol should always follow this invariant in order to keep swapping correctly!

Make a swap

After a pool has liquidity, there are 2 functions users can call to swap tokens in the pool.

- `swapExactInput`
- `swapExactOutput`

A user can either choose exactly how much to input (ie: I want to use 10 USDC to get however much WETH the market says it is), or they can choose exactly how much they want to get out (ie: I want to get 10 WETH from however much USDC the market says it is.

This codebase is based loosely on Uniswap v1

Disclaimer

Jack Landon 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
- In 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

Overall, the protocol works decently and there are no major issues found that require significant reconsideration. All of the issues found are fixable and can be resolved with a few changes to the codebase, whilst still delivering on the protocol's core functionality.

The main issues found are related to the core functionality of the protocol, and the invariant that the protocol is based on. The protocol is based on the `x * y = k` invariant, where `x` is the balance of the `poolToken`, `y` is the balance of `WETH`, and `k` is the constant product of the 2 balances.

This means that when the balances of the 2 tokens change, the product of the 2 balances must remain constant. The issues found are related to the breaking of this invariant, which can lead to the protocol funds being drained from the pool.

Issues found

Severity	Number of Issues Found
HIGH	4
MEDIUM	2

Severity	Number of Issues Found
LOW	2
INFORMATIONAL	9
Gas	1
TOTAL	18

Findings

High Severity

[H-1] An incorrect number being used in the `TSwapPool::getInputAmountBasedOnOutput` function causes users to be charged too high of a fee when calling the `TSwapPool::swapExactOutput` function.

Description: The `TSwapPool::swapExactOutput` function uses the `TSwapPool::getInputAmountBasedOnOutput` utility function to get the required input amount. This `TSwapPool::getInputAmountBasedOnOutput` function uses an incorrect number in it's calculation in the numerator, causing the input amount to be higher than necessary:

▶ Code

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

As the fee amount is 0.3% (0.003), the scalar should be `1_000` instead of `10_000` To calculate the net input amount, the function should return (grossFullAmount - feeAmount). 997 / 10,000 = 0.9997 997 / 1,000 = 0.997

Impact: Protocol takes more fees from users than expected when calling a `TSwapPool::swapExactOutput` function.

Proof of Concept: We can add a test to `TSwapPool.t.sol` to check if the actual input amount taken out > expected input amount taken out:

1. Add liquidity from the `liquidityProvider` to the pool.
2. Start a prank on the `user`.
3. Mint tokens for the `user`.
4. Get the inputBalance of the user *before* the swap
5. Call the `TSwapPool::swapExactOutput` function.
6. Get the inputBalance of the user *after* the swap
7. Assert that the `inputAmount` > `expectedInputAmount`
8. Double-check that the `getInputAmountBasedOnOutputResult` function is the issue by asserting that this output > `expectedInputAmount`

▶ Code

```
    function testIncorrectFeeOnSwapExactOutput() public {
        vm.startPrank(liquidityProvider);
        poolToken.mint(liquidityProvider, 100000e18);
        weth.mint(liquidityProvider, 100000e18);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(user);
        poolToken.mint(user, 100000e18);
        weth.mint(user, 100000e18);

        uint256 userTokenBalanceBefore = poolToken.balanceOf(user);

        uint256 exactOutputAmount = 9e18;
        uint256 inputReservesBefore = poolToken.balanceOf(address(pool));
        uint256 outputReservesBefore = weth.balanceOf(address(pool));

        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, exactOutputAmount,
uint64(block.timestamp));
        vm.stopPrank();

        uint256 userTokenBalanceAfter = poolToken.balanceOf(user);

        uint256 actualChangeInUserTokenBalance = userTokenBalanceBefore -
userTokenBalanceAfter;
        uint256 expectedChangeInUserTokenBalance = ((inputReservesBefore *
exactOutputAmount) * 1000) / ((outputReservesBefore - exactOutputAmount) *
997);

        assert(actualChangeInUserTokenBalance >
expectedChangeInUserTokenBalance);

        uint256 getInputAmountBasedOnOutputResult =
pool.getInputAmountBasedOnOutput(exactOutputAmount, inputReservesBefore,
outputReservesBefore);
        assertEq(getInputAmountBasedOnOutputResult,
actualChangeInUserTokenBalance);
        assert(getInputAmountBasedOnOutputResult >
expectedChangeInUserTokenBalance);
    }
```

Recommended Mitigation: Change the `10000` value to `1000` in the `TSwapPool:getInputAmountBasedOnOutput` function:

```diff
    function getInputAmountBasedOnOutput(
        .
        .
        .
    )
        .
        .
        .

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

    }
```

When we mutate the contract with this change, and update the test outlined in the Proof of Concept to make all `assert`s to `assertEq`, the test passes, proving this is the issue.

[H-2] The lack of a `maxInputAmount` parameter in the `TSwapPool::swapExactOutput` function means that users can be charged significantly more than they intended

Description: If there is no way to set a maximum input amount on the `swapExactOutput` function, attackers could manipulate the pool before their transaction has gone through to drain all of the liquidity from the pool and give the user a much worse rate, and the user has no control over how many funds gets taken out of their balance.

Impact: Impact: HIGH Likelihood: HIGH

Without a check, users can get a much worse rate than when they initiate the transaction. An attacker simply needs to pay more gas to front-run the user's transaction and drain the pool, and then sell them back the liquidity at the new (more expensive) rate.

Proof of Concept: This attack reflects a sandwich attack.

1. Price of USDC -> WETH price is 1,000 USDC -> 1 WETH

2. User wants to get 1 WETH, so they call the `TSwapPool::swapExactOutput` function excpecting it will be ~1,000 USDC.
3. An attacker observes this transaction in the mempool, and puts in a transaction with more gas to have their transaction fulfilled before the user.
4. The attacker's function take out a flash loan of 200,000 USDC to buy a lot of the WETH reserves in the pool, effectively making the price of WETH go up 10x to 10,000 USDC -> 1 WETH.
5. The user's transaction goes through, and they get 1 WETH, but they pay 10,000 USDC for it.
6. In the same block, after the user's transaction, the attacker sells the weth back for 210,000 USDC (excluding fees), and makes good on the USDC flash loan, Making a profit equal to the value of the user's loss, less loan fees + trading fees + gas fees.
7. The effect is that the user sent 10,000 USDC to the pool instead of the expected 1,000 USDC in order to get the 1 WETH.
8. This doesn't have to be a malicious attack and can happen naturally - in a volatile market, the user has no control over how much they are charged for a hard-coded amount of output.

Recommended Mitigation: Copy the similar process to what is done for the slippage protection in the `TSwapPool::swapExactInput` function:

1. Create a custom error in the `TSwapPool` contract to revert if the input amount is too high.
2. Add a `maxInputAmount` parameter to the `swapExactOutput` function.
3. Add a check to ensure that the `inputAmount` is less than or equal to the `maxInputAmount`.

```
    error TSwapPool__InputAmountTooHigh(uint256 inputAmount, uint256
maxInputAmount);
    .
    .
    .

    function swapExactOutput(
        IERC20 inputToken,
+       uint256 maxInputAmount,
        IERC20 outputToken,
        uint256 outputAmount,
        uint64 deadline
    ) public revertIfZero(outputAmount) revertIfDeadlinePassed(deadline)
returns (uint256 inputAmount) {
        .
        .
        .
        inputAmount = getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

+       if (inputAmount > maxInputAmount) {
+           revert TSwapPool__InputAmountTooHigh(inputAmount,
maxInputAmount);
+       }
        .
        .
        .
    }
```

[H-3] The `TSwapPool::sellPoolTokens` function calls the wrong function for deciding to sell an inputted amount of `poolTokenAmount`, causing unexpected behavior for the user.

Description: When calling the `sellPoolTokens`, a user will expect to input some amount of `poolTokens` to sell at the given rate for `WETH`.

The `sellPoolTokens` function however, calls the `TSwapPool::swapExactOutput`, which sets the expected `WETH` output amount to be the inputted amount of `poolTokenAmount` they wish to sell of the pool tokens.

```
    /**
     * @notice wrapper function to facilitate users selling pool tokens in
exchange of WETH
     * @param poolTokenAmount amount of pool tokens to sell
     * @return wethAmount amount of WETH received by caller
     */
    function sellPoolTokens(
        uint256 poolTokenAmount
    ) external returns (uint256 wethAmount) {
        return
            swapExactOutput(
                i_poolToken,
                i_wethToken,
                poolTokenAmount,
                uint64(block.timestamp)
            );
    }
```

Impact: Users will swap the wrong amount of tokens than specified, which is a severe disruption to the functionality of the protocol. Depending on the rate of the pool, the user may unexpectedly spend much more `poolTokenAmount` than they intended to.

Proof of Concept:

Let's say the rate of the pool is 1 `WETH` for 100 `poolToken`.

1. The user calls `sellPoolTokens` and inputs 100 `poolToken` to sell.
2. The `sellPoolTokens` function calls `swapExactOutput` with the `outputAmount` set to 100 (WETH).
3. The intention of the user is to buy 1 `WETH` with their 100 `poolToken`, however what's happening is they're buying 100 `WETH` with an uncapped amount of `poolTokenAmount` as input.
4. The user spends `1000000 poolToken` to buy `100 WETH` instead of the expected `100 poolToken` for `1 WETH`.

In `TSwapPool.t.sol`, add a test to measure the discrepancy between the expected output and input of the `sellPoolTokens` function:

In this proof of code, we assert that the `poolTokenAmount` input we expect to sell is not equal to the `poolTokenAmount` we actually sell.

This would also revert if `WETH` allowance isn't set up for the pool contract.

```
    function testSellPoolTokens() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(user);
        poolToken.mint(user, 100000e18);
        weth.mint(user, 100000e18);

        uint256 poolTokenAmount = 10e18;
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);

        uint256 poolTokenBalanceBeforeSell = poolToken.balanceOf(user);
        pool.sellPoolTokens(poolTokenAmount);

        uint256 poolTokenBalanceAfterSell = poolToken.balanceOf(user);

        uint256 expectedPoolTokenBalanceAfterSell =
poolTokenBalanceBeforeSell - poolTokenAmount;

        assertNotEq(poolTokenBalanceAfterSell,
expectedPoolTokenBalanceAfterSell);
        vm.stopPrank();
    }
```

Recommended Mitigation: Change the `swapExactOutput` to be the `swapExactInput` in the `sellPoolTokens` function:

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

This way, the user gets to control how much `poolTokenAmount` they want to sell, and a minimum amount of `WETH` they get back.

[H-4] `TSwapPool::_swap` function breaks the `x * y = k` invariant by offering extra tokens every 10 swaps

Description: Over the lifecycle of the `TSwapPool` contract, the `x * y = k` invariant must strictly hold where:

- `x` is the balance of the `poolToken`,
- `y` is the balance of `WETH`, and
- `k` is the constant product of the 2 balances.

This means that when the balances of the 2 tokens change, the product of the 2 balances must remain constant. By offering bonus tokens every 10 swaps, this rule invariant is broken.

This code block from the `TSwapPool::_swap` function 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: By breaking this invariant, over time, the protocol funds will be drained from the pool.

An attack could drain the funds from the pool by doing many swaps (wash trading), where each trade cancels out the other and takes all of the liquidity in the form of bonus tokens.

Proof of Concept:

1. An attacker swaps 10 times and collects the incentive `1_000_000_000_000_000_000` tokens.
2. They continue to attack until the pool is drained of all liquidity by receiving `1_000_000_000_000_000_000` every 10 swaps.

▶ Proof Of Code

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

```
function testBrokenInvariant() 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.mint(user, 100000e18);
        weth.mint(user, 100000e18);

        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); //
Negative because we're losing weth

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

        // Actual
        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);

        assertEq(actualDeltaY, expectedDeltaY);
    }
```

Recommended Mitigation: In the `_swap` function, remove this feature: `* @dev Every 10 swaps, we give the caller an extra token as an extra incentive to keep trading on T-Swap.`

In the code, it can be done like this:

```
function _swap(
        .
        .
        .
    ) private {
        .
        .
```

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

This will keep the invariant in check.

Alternatively, the swap incentive can be kept, but should be dished out in the same way that fees are distributed.

Medium Severity

[M-1] The `TSwapPool::deposit` function has an unused `deadline` parameter, which is misleading and could cause transactions to proceed after the intended deadline

Description: The `deadline` parameter is not used in the `deposit` function. When the user inputs some `deadline` value in the `TSwapPool::deposit` function which isn't honored, it can have a significant unintended consequenses. An example is that between the time that the user calls the `TSwapPool::deposit` function and the time in which is completes, the market conditions may have changed and they may get an unfavorable rate. Beyond this, MEV attacks are possible if the `deadline` is not honored.

Impact: Transactions could be sent when market conditions when market confitions are unfavourable, even when adding a `deadline`. Impact: MEDIUM Likelihood: MEDIUM

Proof of Concept: The `deadline` parameter is unused:

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

Recommended Mitigation: The function should be changed to consider the inputted `deadline`, or removed entirely to avoid confusion.

Add the `revertIfDeadlinePassed` modifier to the `TSwapPool::deposit` function.

```
        function deposit(
            uint256 wethToDeposit,
```

```diff
        uint256 minimumLiquidityTokensToMint, // LP tokens -> if empty,
pick a number (eg 17) => 17 tokens == 100% (q? what if we set the value to
type uint256.max())
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
+       revertIfDeadlinePassed(deadline)
        returns (uint256 liquidityTokensToMint)
    {...}
```

This is in line with the `TSwapPool:deposit` natspec (`@param deadline The deadline for the transaction to be completed by`), however if this goes against the intended functionality, the `deadline` parameter should be removed from the function:

```diff
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint, // LP tokens -> if empty,
pick a number (eg 17) => 17 tokens == 100% (q? what if we set the value to
type uint256.max())
        uint256 maximumPoolTokensToDeposit,
-       uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {...}
```

[M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol invariant

Description: ERC-20's which change the balance of the `poolToken` under the hood can break the `x * y = k` invariant.

Impact: In the example of an ERC-777, when a `_swap` function is called, an attacker can received the `poolTokens` to a contract which has an implementation of a `receive()` function, which calls the pools `_swap` function again, effectively draining the liquidity from the pool. by sending the pool tokens to a contract that doesn't have the `receive` function, which will revert the transaction and drain the pool.

Proof of Concept:

1. If an attacker receives some amount of `poolToken` which is an ERC-777, they can execute anythin before the finalization of the transaction.
2. The attacker calls `TSwapPool::swapExactInput` with `weth` and `poolToken` as the input and output tokens, respectively.
3. The `_swap` function calls `outputToken.safeTransfer(msg.sender, outputAmount);`, to a contract with a `receive()` function which calls the `_swap` function again.

Recommended Mitigation: Add a locking function to the `_swap` function to prevent re-entrancy. A good way to do this is with OpenZeppelin's `ReentrancyGuard` contract.

```diff
+   import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
    .
    .
    .
    function _swap(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 outputAmount
-   ) private {
+   ) private nonReentrant() {
    .
    .
    .
}
```

Low Severity

[L-1] Emitted `LiquidityAdded` event in `TSwapPool::_addLiquidityMintAndTransfer` function has parameters in the wrong order causing incorrect data to be emitted

Description: The `LiquidityAdded` event is declared as follows:

```
    event LiquidityAdded(
        address indexed liquidityProvider,
        uint256 wethDeposited,
        uint256 poolTokensDeposited
    );
```

When the `LiquidityAdded` is emitted in the function, it is written like this:

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

It's clear that the `poolTokensToDeposit` and `wethToDeposit` values are in the wrong order.

Impact: This won't have a direct impact on the functionality or logic of the contract, however off-chain tooling uses this to parse events, and will therefore assume the wrong data.

This can have a greater negative impact if a contract relies on this event information for the correct functioning of their contract.

Recommended Mitigation: In the `TSwapPool::_addLiquidityMintAndTransfer` function, change the `LiquidityAdded` event to emit the correct values:

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

[L-2] The `TSwapPool::swapExactInput` function returns an `output` value that isn't updated, meaning the output of the function is misleading

Description: The `TSwapPool::swapExactInput` function declares the returned `output` value:

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
@>      returns (uint256 output)
    {...}
```

However, in the implementation of the function, the `output` value is not updated, nor explicitly returned:

▶ Code

```
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);

        // Nothing returned here
    }
```

Impact: As the contract doesn't use the output of this function, the logic or core functionality of the contract isn't affected. This can be misleading to users or consumers of the contract if they expect an output value to be returned.

Proof of Concept:

We can make a test to prove that the returned value of the `swapExactInput` function returns an incorrect value:

1. First the `liquidityProvider` adds liquidity to the pool.
2. Then we start a prank on the `user`.
3. Mint tokens for the `user`.
4. Get the output (WETH) balance of the user *before* the swap.
5. Get the `expectedReturnOutputAmount` based on the input amount.
6. Call the `TSwapPool::swapExactInput` function.
7. Get the output (WETH) balance of the user *after* the swap.
8. Assert that the `outputBalanceAfter - outputBalanceBefore` is equal to the `expectedReturnOutputAmount`, which means the `getOutputAmountBasedOnInput` is correct
9. Assert that the `returnedOutputAmount` is not equal to the `expectedReturnOutputAmount`, which means that the returned value of the `swapExactInput` is incorrect.

```
function testNoReturnOfSwapExactInput() public {
        vm.startPrank(liquidityProvider);
        poolToken.mint(liquidityProvider, 100000e18);
        weth.mint(liquidityProvider, 100000e18);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        vm.startPrank(user);
        poolToken.mint(user, 100000e18);

        uint256 expected = 9e18;
        uint256 inputAmount = 10e18;
        uint256 inputReserves = poolToken.balanceOf(address(pool));
        uint256 outputReserves = weth.balanceOf(address(pool));

        uint256 outputBalanceBefore = weth.balanceOf(user);
        poolToken.approve(address(pool), 10e18);
        uint256 expectedReturnOutputAmount =
pool.getOutputAmountBasedOnInput(
            inputAmount,
            inputReserves,
            outputReserves
        );

        uint256 returnedOutputAmount = pool.swapExactInput(poolToken,
inputAmount, weth, expected, uint64(block.timestamp));

        uint256 outputBalanceAfter = weth.balanceOf(user);

        console.log(returnedOutputAmount);
        console.log(outputBalanceAfter - outputBalanceBefore);

        assertEq(outputBalanceAfter - outputBalanceBefore,
expectedReturnOutputAmount);
        assertNotEq(returnedOutputAmount, expectedReturnOutputAmount);

        vm.stopPrank();
    }
```

The test also logs the return value of the `swapExactInput` function, which is 0. Contrarilly, the actual `outputBalanceAfter - outputBalanceBefore` is `9066108938801491315`.

Recommended Mitigation:

Simply return the output amount of the swap at the end of the function:

```
function swapExactInput(
        .
        .
        .
    )
        .
        .
        .
-       returns (uint256 output)
+       returns (uint256)
    {
        .
        .
        .
        uint256 outputAmount = getOutputAmountBasedOnInput( inputAmount,
inputReserves, outputReserves);
        .
        .
        .
        _swap(inputToken, inputAmount, outputToken, outputAmount);
+       return outputAmount;
    }
```

Gas Optimization

[G-1] In the `TSwapPool::deposit` function, the `poolTokenReserves` value is not used, and can therefore be removed

Description: The `poolTokenReserves` makes an external call to get the balance of the pool contract, and then doesn't use this value for the remainder of the function.

This is unneccessary use of gas, and should therefore be removed.

Recommended change:

```diff
    function deposit(
        .
        .
        .
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        .
        .
        .
        if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this));
-           uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
            .
            .
            .
        } else {
            .
            .
            .
        }
    }
```

Informational

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

Description: An unused error will use more gas for deployment, and can unneccessarily clutter the codebase. As there is no use for the `PoolFactory::PoolFactory__PoolDoesNotExist` it should be removed.

Recommended Change:

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

[I-2] Constructors lack an `address(0)` check in the constructor when setting state/immutable variables

Description: It's important that certain `address` values are not `address(0)` for the contracts to function correctly. The `address(0)` check should be added to the constructor when setting these variables.

This affects the following variables:

- `PoolFactory::i_wethToken`
- `TSwapPool::i_wethToken`
- `TSwapPool::i_poolToken`

Recommended Change:

In the `PoolFactory` contract:

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

In the `TSwapPool` contract:

```diff
    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-3] When setting the `liquidityTokenSymbol` in the `PoolFactory::createPool` function, the token `name` is used instead of the symbol

Description: In the `PoolFactory::createPool` function, the `liquidityTokenSymbol` is set by concattenating the token name onto the `ts` string. Token symbols are usually short indentifiers. The token name could be long and not suitable for a symbol. Instead, the token's symbol should be used.

Recommended change:

```diff
    function createPool(address tokenAddress) external returns (address) {
        .
        .
        .
        string memory liquidityTokenName = string.concat("T-Swap ",
IERC20(tokenAddress).name());
-       string memory liquidityTokenSymbol = string.concat("ts",
IERC20(tokenAddress).name());
+       string memory liquidityTokenSymbol = string.concat("ts",
IERC20(tokenAddress).symbol());
        .
        .
        .
        return address(tPool);
    }
```

[I-4] If the `IERC20(tokenAddress).name()` call in `PoolFactory::createPool` fails, the function will revert, and a pool won't be able to be created for tokens with no valid return for `name()`

Description: If the `IERC20(tokenAddress).name()` call fails, the `createPool` function will revert, and a pool won't be able to be created for tokens with no valid return for `name()`. This could be due to a token not implementing the `name()` function, or the token not being a valid ERC20 token.

Recommended Mitigation: Add a fallback for tokens that do not have a name, and revert with an error message.

```diff
    function createPool(address tokenAddress) external returns (address) {
        .
        .
        .
-       string memory liquidityTokenName = string.concat("T-Swap ",
IERC20(tokenAddress).name());
+       string memory liquidityTokenName = string.concat("T-Swap ",
IERC20(tokenAddress).name());
+       if (bytes(liquidityTokenName).length == 0) revert("Token does not
have a name");
        .
        .
        .
        return address(tPool);
    }
```

[I-5] Event is missing `indexed` fields in `TSwapPool`.

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.

The affected events are:

- `PoolFactory::PoolCreated`,
- `TSwapPool::LiquidityAdded`,
- `TSwapPool::LiquidityRemoved`, and
- `TSwapPool::Swap`

Recommended change:

In the `PoolFactory` Contract:

```diff
-   event PoolCreated(address tokenAddress, address poolAddress);
+   event PoolCreated(address indexed tokenAddress, address indexed
poolAddress);
```

In the `TSwapPool` Contract:

```diff
    event LiquidityAdded(
        address indexed liquidityProvider,
-       uint256 wethDeposited,
+       uint256 indexed wethDeposited,
-       uint256 poolTokensDeposited
+       uint256 indexed poolTokensDeposited
    );
    event LiquidityRemoved(
        address indexed liquidityProvider,
-       uint256 wethWithdrawn,
+       uint256 indexed wethWithdrawn,
-       uint256 poolTokensWithdrawn
+       uint256 indexed poolTokensWithdrawn
    );
    event Swap(
        address indexed swapper,
-       IERC20 tokenIn,
+       IERC20 indexed tokenIn,
        uint256 amountTokenIn,
-       IERC20 tokenOut,
+       IERC20 indexed tokenOut,
        uint256 amountTokenOut
    );
```

[I-6] The `TSwapPool::deposit` function doesn't follow Checks-Effect-Interactions pattern when `totalLiquidityTokenSupply == 0`

Description: At the end of the `TSwapPool::deposit` function when there are no issued liquidity tokens, the contract mints the tokens for the user, and *then* updates the `liquidityTokensToMint` value to be returned.

Impact: As the `liquidityTokensToMint` isn't a state variable, this isn't a critical issue, but it's best practice to follow the Checks-Effects-Interactions pattern.

Recommended change:

Update the `liquidityTokensToMint` value before calling the `_addLiquidityMintAndTransfer` function, and then return `liquidityTokensToMint`.

```diff
    function deposit(
        uint256 wethToDeposit,
        .
        .
        .
    )
        .
        .
        .
    {
        .
        .
        .
        if (totalLiquidityTokenSupply() > 0) {
            .
            .
            .
        } else {
+           liquidityTokensToMint = wethToDeposit;
            _addLiquidityMintAndTransfer(
                wethToDeposit,
                maximumPoolTokensToDeposit,
                wethToDeposit
            );
-           liquidityTokensToMint = wethToDeposit;
+           return liquidityTokensToMint;
        }
    }
```

[I-7] "Magic Numbers" are used in the `TSwapPool` contract, which can hinder code readability and may cause errors

Description: When using numbers directly in the code, it's better to define them as a named constant, and used them in the code like this. This makes the code more readable and easier to maintain.

In the `TSwapPool` contract, "magic numbers" are used in the following functions:

- `TSwapPool::getOutputAmountBasedOnInput` function

```
        uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * 1000) +
inputAmountMinusFee;
```

- `TSwapPool::getInputAmountBasedOnOutput` function

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

When reading the code, it's hard to know what the `997`, `1000`, and `10000` values represent. It's better to define these values as named constants.

It's likely that because magic numbers are used, `10000` was incorrectly used, where it is supposed to be `1000`.

Recommended Mitigation:

In the `TSwapPool` contract, define named constants for the magic numbers:

```
+   uint256 private constant FEE_PRECISION = 1000;
+   uint256 private constant AMOUNT_NET_OF_FEE = 997;
```

Then use these constants in the `TSwapPool` contract.

- `TSwapPool::getOutputAmountBasedOnInput` function:

```
-       uint256 inputAmountMinusFee = inputAmount * 997;
+       uint256 inputAmountMinusFee = inputAmount * AMOUNT_NET_OF_FEE;
        uint256 numerator = inputAmountMinusFee * outputReserves;
-       uint256 denominator = (inputReserves * 1000) +
inputAmountMinusFee;
+       uint256 denominator = (inputReserves * FEE_PRECISION) +
inputAmountMinusFee;
```

- `TSwapPool::getInputAmountBasedOnOutput` function

```
        return
-            ((inputReserves * outputAmount) * 10000) / ((outputReserves -
outputAmount) * 997);
+            ((inputReserves * outputAmount) * FEE_PRECISION) /
((outputReserves - outputAmount) * AMOUNT_NET_OF_FEE);
```

[I-9] The `TSwapPool::swapExactInput` and `TSwapPool::totalLiquidityTokenSupply` functions visibility are set to `public` when they aren't used in the in the contract, causing higher gas fees and unneccessary visibility lenience

Description: A function that doesn't get used in the contract should be marked as `external`. This will save on gas each time it is called, and prevent potential vulnerabilities as you don't want the `TSwapPool` function to ever call it.

Impact: Gas savings and more secure code.

Recommended Mitigation:

1. In the `TSwapPool::swapExactInput` function, change the `public` keyword to `external`:

```diff
    function swapExactInput(
        .
        .
        .
    )
-       public
+       external
        .
        .
        .
    {...}
```

2. In the `TSwapPool::totalLiquidityTokenSupply` function, change the `public` keyword to `external`:

```diff
-   function totalLiquidityTokenSupply() public view returns (uint256) {
+   function totalLiquidityTokenSupply() external view returns (uint256) {
        return totalSupply();
    }
```

[I-10] The `TSwapPool::swapExactInput` function has no natspec, causing people to guess the purpose and inputs of the function

Description: It's important to be as explicit as possible when describing the purpose and logic of the function. The `TSwapPool::swapExactInput` function has no comments at all, and to assess it, one must gather information from the `TSwapPool::swapExactOutput` function and make assumptions based on the natspec for this.