TSwap protocol Audit Report

Version 1.0

Fawarano

October 2, 2025

TSwap protocol Audit Report

Fawarano

October 02, 2025

Prepared by: Fawarano Lead Auditors:

- Fawarano

Table of Contents

    * [H-4] On `TSwapPool::swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`
  - Medium
    * [M-1] Unused `deadline` Parameter in `TSwapPool::deposit` causing transactions to complete even after the deadline is passed
  - Low
    * [L-1] Incorrect Parameter Order in `TSwapPool::LiquidityAdded` Event causing event to emit incorrect information
    * [L-2] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given
  - Informationals
    * [I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed
    * [I-2] Missing Zero Address Validation in Constructor
    * [I-3] `PoolFadctory::createPool` should use `.symbol` instead of `.name`
    * [I-4] Emission of Constant Value `MINIMUM_WETH_LIQUIDITY` Increases Gas Costs
    * [I-5] Unused Variable Leads to Wasted Gas
    * [I-6] Non-CEI Pattern in `TSwapPool::_addLiquidityMintAndTransfer`

Protocol Summary

his 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

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.

Disclaimer

The Fawarano auditor 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

- Commit Hash: e643a8d4c2c802490976b538dd009b351b1c8dda

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

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

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many fees from users

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user should deposit given an amount of token of outuput tokens. However, the function miscalculates the resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Recommended Mitigation:

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

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

Description: The function `TSwapPool::swapExactOutput` calculates the required `inputAmount` for a desired `outputAmount`, but does not include a maximum input amount check. This omission means users cannot protect themselves from excessive slippage. If reserves shift between the transaction submission and mining, the actual `inputAmount` could be significantly higher than expected, causing users to overpay.

Impact: Users expecting a maximum spend may unknowingly pay excessive amounts if the market conditions change before the transaction processes, resulting in loss of value due to lack of slippage protection

Proof of Concept:

1. The price of 1 WETH righ now is 1,000 USDC
2. User call `swapExactOutput` and is looking for 1 WETH
    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline = whatever
3. The function does not offer a maxInputAmount
4. As the transaction is pending in mempool, the market changes and prices increases 1 WETH is now 10,000 USDC. 10x more than what the user expected
5. The transaction completes, but the user sent to the protocol 10,000 USDC instead of 1,000

Recommended Mitigation: Add a `maxInputAmount` parameter to the function and enforce a check. This ensures swaps fail if slippage exceeds the user's tolerance, protecting users against unexpected losses.

```diff
    function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxInputAmount,
        uint64 deadline
    .
    .

        uint256 outputReserves = outputToken.balanceOf(address(this));

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

        inputAmount = getInputAmountBasedOnOutput(outputAmount,
           inputReserves, outputReserves);

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

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

Description: The sellPoolTokens function incorrectly calls `swapExactOutput` instead of `swapExactInput`. Additionally, the function signature used for `swapExactOutput` does not match the intended parameters, leading to parameter mismatch.

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

Proof of Concept:

Recommended Mitigation: Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this also require to add a new parameter to the function (ie `minWethToReceive` to be passed to `swapExactInput`)

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

[H-4] On `TSwapPool::swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`

Description: The protocol follows a strict invariant of `x * y = k`. Where:

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

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

The 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.

Proof of Concept:

1. A user swaps 10 times, and collect the extra incentive of `1_000_000_000_000_000_000`
2. That user contines to swap untill all the protocol funds are drained

Place the following into `TSwapPool.t.sol`

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));

          uint256 outputWeth = 1e17;
          int256 startingY = int256(weth.balanceOf(address(pool)));
          int256 expectedDeltaY = int256(-1) * (int256(outputWeth));

          vm.startPrank(user);
          poolToken.mint(user, 100 ether);
          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));
        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 change in the protocol invariant or we should set aside tokens in the same way we do with fees.

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

Medium

[M-1] Unused `deadline` Parameter in `TSwapPool::deposit` causing transactions to complete even after the deadline is passed

Description: The `TSwapPool::deposit` function accepts a `deadline` parameter intended to protect users from transactions being executed later than expected. However, this `deadline` is never checked in the function body. As a result, operations that add liquidity to the pool might be executed at unexpected times,in market conditions where the deposit rate is unfavorable.

This renders the `deadline` parameter useless and may expose users to MEV and front-running attacks, where an attacker can delay a transaction and execute it later under less favorable conditions.

Impact

Impact:

1. User protection bypassed: Users rely on deadlines to ensure their transaction executes promptly or not at all. Since the deadline is not enforced, transactions may succeed long after the intended time.
2. Increased MEV risk: Attackers can intentionally delay user deposits and force execution when token ratios, pool reserves, or gas conditions are unfavorable.
3. Loss of trust in contract guarantees: The parameter gives the impression of safety but offers no protection.

Recommended Mitigation: Add an explicit check to enforce the deadline before executing the deposit logic:

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

Low

[L-1] Incorrect Parameter Order in TSwapPool`::LiquidityAdded` Event causing event to emit incorrect information

Description: The `TSwapPool::LiquidityAdded` event is emitted with parameters in the wrong order. Based on the expected event signature, the correct order should be `(msg.sender, wethToDeposit, poolTokensToDeposit)`. Emitting events with incorrect parameter order can lead to misleading or incorrect on-chain logs.

Impact: Event logs will be misinterpreted by external applications, analytics, or frontends, potentially causing confusion or incorrect reporting of liquidity additions. External tools reading the event will misreport poolTokens as weth and vice versa, leading to inaccurate dashboards or metrics.

Recommended Mitigation: orrect the parameter order when emitting the event. This ensures that event data matches the expected schema and prevents misinterpretation

```
-   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 function swapExactInput declares a return type uint256 output but does not explicitly return a value. Instead, it calls internal logic `_swap` and exits without returning outputAmount. This results in Solidity defaulting to returning 0, which is incorrect and misleading for external callers.

Impact: External integrations relying on the return value will receive 0 instead of the actual swapped output amount, leading to faulty assumptions, incorrect accounting, or broken dApp frontends.

Proof of Concept: A user calls `swapExactInput` expecting it to return the number of tokens received.

Instead, the function always returns 0, regardless of the actual transfer.

Recommended Mitigation: Explicitly return the computed output amount. This ensures that the function behaves as expected and external consumers receive the correct swap result.

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

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

Informationals

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

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

[I-2] Missing Zero Address Validation in Constructor

Description: In the constructor, the contract sets the immutable `i_wethToken` to the address passed by the deployer:

```
constructor(address wethToken) {
@>     i_wethToken = wethToken;
}
```

However, there is no validation to ensure that the provided wethToken address is not the zero address (address(0)). If address(0) is mistakenly passed during deployment, the contract will be permanently configured with an invalid token address. This could break functionality wherever i_wethToken is used (e.g., transfers, approvals, or external calls).

Impact: The contract would be non-functional when interacting with the WETH token (or the expected ERC20).

Although the owner controls deployment and the risk is low (human error), this misconfiguration would be irreversible, since the immutable variable cannot be changed after deployment.

Recommended Mitigation: Add a validation check in the constructor to ensure the provided address is not the zero address:

```
constructor(address wethToken) {
+     require(wethToken != address(0), "Invalid wethToken address");
    i_wethToken = wethToken;
}
```

[I-3] `PoolFadctory::createPool` should use `.symbol` instead of `.name`

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

[I-4] Emission of Constant Value `MINIMUM_WETH_LIQUIDITY` Increases Gas Costs

Description: In the `TSwapPool::deposit` function, the contract reverts with `TSwapPool__WethDepositAmou` `(MINIMUM_WETH_LIQUIDITY, wethToDeposit)` if the deposit amount is below the minimum. The constant MINIMUM_WETH_LIQUIDITY is emitted unnecessarily.

Impact: Unnecessary gas consumption when the revert condition is triggered, since constants don't need to be included in revert data.

Recommended Mitigation:
Emit only the dynamic parameter in the revert reason and rely on the constant defined in the contract.

```
if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
-     revert TSwapPool__WethDepositAmountTooLow(MINIMUM_WETH_LIQUIDITY,
    wethToDeposit);
+     revert TSwapPool__WethDepositAmountTooLow(wethToDeposit);
}
```

[I-5] Unused Variable Leads to Wasted Gas

Description: In the `TSwapPool::deposit` function, the variable `poolTokenReserves` is assigned as `i_poolToken.balanceOf(address(this))` but never used afterwards.

Impact: The unnecessary computation increases gas costs without providing any functional benefit.

Recommended Mitigation: Remove the unused variable assignment to save gas.

[I-6] Non-CEI Pattern in `TSwapPool::_addLiquidityMintAndTransfer`

Description: The code snippet in question performs an external call to _addLiquidityMintAndTransfer before updating the local variable liquidityTokensToMint. While there is no immediate reentrancy vulnerability since liquidityTokensToMint is a local variable (not a state variable), this violates the Checks-Effects-Interactions (CEI) pattern, which is considered best practice to prevent potential reentrancy issues in the future.

Impact: Currently, there is no direct security risk, but failing to follow CEI could lead to bugs or vulnerabilities if the code is modified in the future to use state variables instead of local variables.

External calls before updating variables can lead to unexpected behaviors or reentrancy issues in more complex scenarios.

Recommended Mitigation: Reorder the code to update all effects before making external calls:

```
liquidityTokensToMint = wethToDeposit; // update effect first
_addLiquidityMintAndTransfer(
    wethToDeposit,
    maximumPoolTokensToDeposit,
    wethToDeposit
); // then external call
```

This aligns with the CEI pattern and improves the overall safety of the contract.