TSwap Audit Report

Version 1.0

Sagar Rana

June 13, 2025

TSwap Audit Report

Sagar Rana

June 13, 2025

Prepared by: Sagar Rana Lead Auditors: - Sagar Rana

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). This is a modified fork of the Uniswap V1.

Disclaimer

Sagar Rana 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
Likelihood	High	H	H/M	M
	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

Solc Version: 0.8.20

Chain(s) to deploy contract to: Ethereum

Tokens: Any ERC20 token

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.

Issues found

Severity	Number of issues found
High	3
Medium	3
Low	4
Informational	7
Gas	2
Total	19

Findings

High

[H-1] `TSwapPool::deposit()` is missing deadline check causing transactions to complete even after deadline

Description The `deposit()` function accepts a `deadline` paramter which according to the natspec is the deadline for the transaction to be completed by, however this paramaeter is never used.

Impact Operations that add liquidity may be executed in unexpected times, in market conditions where deposit rates are unfavourable.

Proof of Concepts The `deadline` paramater is unused, also shown by the compiler:

```
Compiler run successful with warnings:
Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
   --> src/TSwapPool.sol:100:9:
    |
100 |         uint64 deadline
    |         ^^^^^^^^^^^^^^^
```

Recommended mitigation Make the following change to the function:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
```

```
5       uint64 deadline
6   )
7       external
8   +   revertIfDeadlinePassed(deadline)
9       revertIfZero(wethToDeposit)
10      returns (uint256 liquidityTokensToMint)
11  {
12      if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
13          revert TSwapPool__WethDepositAmountTooLow(
                MINIMUM_WETH_LIQUIDITY, wethToDeposit);
14      }
15      if (totalLiquidityTokenSupply() > 0) {
16          uint256 wethReserves = i_wethToken.balanceOf(address(this));
```

[H-2] Typo during fees calculation makes the fees as high as `90.03%`

Description `TSwapPool::getInputAmountBasedOnOutput()` has a fees mechanism which is supposed to calculate 0.3% fees during a swap by calculating it as `997/1000`. But due there has been a typo which calculates the fees as `997/10000`

Impact Due to the error, the fees for the swap is calculated as 90.03% instead of 0.3%

Proof of Concepts

```
1  // Expected fee calculation
2  100 - [(997 / 1_000) * 100] = 100 - (0.997 * 100) = 100 - 99.7 = 0.3%
3
4  // Actual fee calculation
5  100 - [(997 / 10_000) * 100] = 100 - (0.0997 * 100) = 100 - 9.97 =
       90.03%
```

It is clear in the below code:

```
1  function getInputAmountBasedOnOutput(
2      uint256 outputAmount,
3      uint256 inputReserves,
4      uint256 outputReserves
5  )
6      public
7      pure
8      revertIfZero(outputAmount)
9      revertIfZero(outputReserves)
10     returns (uint256 inputAmount)
11 {
12 @>   return ((inputReserves * outputAmount) * 10000) / ((outputReserves
       - outputAmount) * 997);
13 }
```

Recommended mitigation

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

[H-3] In `TSwapPool::_swap()` the extra tokens given to users break the invariant

Description The protocol follows a strict invariant of `x * y = k` where: - `x`: balance of pool token - `y`: balance of WETH token - `k`: constant product of two balances Thus, after every swap the `k` must remain constant and not change. However, in the `_swap()` function, for swapping incentives the users are rewarded with `1e18` every 10 swaps.

Impact This extra token rewarded to user every 10 swaps breaks the core invariant of the protocol.

Proof of Concepts Add the following testcase to the `TSwapPool` test suite:

```
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 = 1e8;

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

    vm.startPrank(user);

    weth.approve(address(pool), type(uint256).max);
    poolToken.approve(address(pool), type(uint256).max);
    poolToken.mint(user, 100e18);
    for (uint8 i; i < 9; i++) {
```

```
19          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
               timestamp));
20      }
21
22      vm.stopPrank();
23
24      int256 endingY = int256(weth.balanceOf(address(pool)));
25      int256 actualDeltaY = endingY - startingY;
26
27      assertEq(actualDeltaY, expectedDeltaY);
28  }
```

Recommended mitigation

```
1   function _swap(IERC20 inputToken, uint256 inputAmount, IERC20
       outputToken, uint256 outputAmount) private {
2       if (_isUnknown(inputToken) || _isUnknown(outputToken) || inputToken
            == outputToken) {
3           revert TSwapPool__InvalidToken();
4       }
5
6   -   swap_count++;
7   -   if (swap_count >= SWAP_COUNT_MAX) {
8   -       swap_count = 0;
9   -       outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000)
        ;
10  -   }
11      emit Swap(msg.sender, inputToken, inputAmount, outputToken,
           outputAmount);
12
13      inputToken.safeTransferFrom(msg.sender, address(this), inputAmount)
           ;
14      outputToken.safeTransfer(msg.sender, outputAmount);
15  }
```

Medium

[M-1] No slippage check in `TSwapPool::swapExactOutput()`

Description There is no `maxInputAmount` param in `TSwapPool::swapExactOutput()` to check for slippage. A check similar to the slippage check in `TSwapPool::swapExactInput()` should be done

Impact The swap may execute in unfavourable market conditions and drain too many tokens from user's balance

Proof of Concepts 1. The price of WETH right now is 1,000 USDC 2. User inputs a `swapExactOutput` looking for 1 WETH 1. `inputToken` = USDC 2. `outputToken` =

WETH 3. `outputAmount` = 1 4. `deadline` = anything 3. The function does not offer a `maxInputAmount` 4. As the transaction is pending in the mempool, the market condition changes and the new price of 1 WETH is now 2,000 USDC 5. User, who was expecting to be deducted 1,000 USDC now gets deducted 2,000 USDC, double of what they were expecting.

Recommended mitigation

```
+error TSwapPool__OutputTooLHigh(uint256 actual, uint256 max);
.
.
.
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, outputReserves);

+   if (inputAmount < maxInputAmount    ) {
+       revert TSwapPool__OutputTooHigh(inputAmount, maxInputAmount);
+   }
+
    _swap(inputToken, inputAmount, outputToken, outputAmount);
}
```

[M-2] `TSwapPool::sellPoolTokens()` uses the `swapExactOutput()` instead of `swapExactInput()` for selling tokens, unable to input number of tokens to sell

Description The `sellPoolTokens()` function is expected to receive the `poolTokenAmount` parameter for choosing the number of tokens to sell. The `swapExactInput()` function should be used as we know the amount of tokens we want to sell. But instead it uses the `swapExactOutput()` function which requires us to specify the number of output tokens we want to receive.

Impact Unfavourable market conditions may drain too many poolTokens from the contract in unfavourable market conditions

Proof of Concepts - The signature of `sellPoolTokens()` is `sellPoolTokens(uint256 poolTokenAmount)external returns (uint256 wethAmount)`, suggesting that we know the amount of tokens to sell - We are calling the `swapExactOutput()` function with parameters `swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount, uint64(block.timestamp))` - Signature of `swapExactOutput()` is `swapExactOutput (IERC20 inputToken, IERC20 outputToken, uint256 outputAmount, uint64 deadline)` - Thus, our `poolTokenAmount` is passed as the `outputAmount` when it should be the `inputAmount`. This operation will buy poolTokens instead of sell them!

Recommended mitigation

```
1 function sellPoolTokens(uint256 poolTokenAmount) external returns (
    uint256 wethAmount) {
2 -    return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount,
    uint64(block.timestamp));
3 +    uint256 minOutputAmount = getOutputAmountBasedOnInput(
    poolTokenAmount, i_poolToken.balanceOf(address(this)), i_wethToken.
    balanceOf(address(this)));
4 +    return swapExactInput(i_poolToken, poolTokenAmount, i_wethToken,
    minOutputAmount, uint64(block.timestamp));
5 }
```

[M-3] Rebase, fee-on-transfer and ERC777 tokens break protocol invariant

Description - Rebase tokens: A rebase ERC20 token automatically adjusts its total supply over time. Instead of transferring tokens to or from holders during supply changes, a rebase function in the contract proportionally changes everyone's balance. - Fee-on-Transfer tokens: Some tokens as an incentive for doing transactions provide rewards to the owner every few transactions. This reward can break invariants across many protocols - ERC777 tokens: ERC777 tokens can allow reentrancy attacks and this can drain all the funds from a protocol

Impact Due to not following the ERC20 standards these types of tokens can cause unexpected behaviours while transactions.

Proof of Concepts - Rebase functions: Since these tokens do not transfer actual tokens but rather a frctions of ownership, TSwap contract swaps will behave unexpectedly. - Fee-on-transfers: The rewarded tokens will cause invariant breaks - ERC777 tokens: These tokens can cause reentrancy attacks and drain liquidity pools.

Recommended mitigation 1. Refactor codebase to consider all types of ERC20s 2. Use `SafeERC` contract from OpenZeppelin

Low

[L-1] No check for equality against `i_wethToken` in `PoolFactory::createPool()` can create WETH-WETH pool

Description The `PoolFactory::createPool()` function does not have any checks to ensure that `tokenAddress` and `i_wethToken` are not the same

Impact Anybody can call the `createPool()` function with the WETH token address and create a WETH pool which will be invalid

Proof of Concepts Add the following test case to the PoolFactory test suite:

```
function test_createWethPool() public {
    PoolFactory newFactory = new PoolFactory(address(mockWeth));
    address poolAddress = newFactory.createPool(address(mockWeth));
    TSwapPool pool = TSwapPool(poolAddress);

    assertEq(pool.getPoolToken(), pool.getWeth());
}
```

Then run the following command:

```
forge test --mt test_createWethPool
```

Recommended mitigation

```
function createPool(address tokenAddress) external returns (address) {
+   assert(tokenAddress != i_wethToken);
    if (s_pools[tokenAddress] != address(0)) {
        revert PoolFactory__PoolAlreadyExists(tokenAddress);
    }
```

[L-2] No zero address checks in `TSwapPool` constructor

Description There are no checks to ensure that the token address passed into constructor for pool creation is not a zero address

Impact A pool of invalid token may be created

Proof of Concepts Add the following test case to the PoolFactory test suite:

```
function test_createZeroAddressPool() public {
    PoolFactory newFactory = new PoolFactory(address(mockWeth));
    address poolAddress = newFactory.createPool(address(0));
    TSwapPool pool = TSwapPool(poolAddress);

    assertEq(pool.getPoolToken(), address(0));
```

```
7 }
```

Then run the following command:

```
forge test --mt test_createZeroAddressPool
```

Recommended mitigation

```
constructor(
    address poolToken,
    address wethToken,
    string memory liquidityTokenName,
    string memory liquidityTokenSymbol
)
    ERC20(liquidityTokenName, liquidityTokenSymbol)
{
+   assert((wethToken != address(0)) && (poolToken != address(0)));
    i_wethToken = IERC20(wethToken);
    i_poolToken = IERC20(poolToken);
}
```

[L-3] No check for inequality of `wethToken` and `poolToken` in `TSwapPool::constructor()`

Description The `TSwapPool::constructor()` function does not have any checks to ensure that `wethToken` and `poolToken` are not the same

Impact Anybody can call the `constructor()` function with the WETH token address and create a WETH pool which will be invalid

Proof of Concepts Add the following test case to the PoolFactory test suite:

```
function test_createWethPoolTSwapPool() public {
    PoolFactory newFactory = new PoolFactory(address(mockWeth));
    address poolAddress = newFactory.createPool(address(mockWeth));
    TSwapPool pool = TSwapPool(poolAddress);

    assertEq(pool.getPoolToken(), pool.getWeth());
}
```

Then run the following command:

```
forge test --mt test_createWethPoolTSwapPool
```

Recommended mitigation

```
constructor(
    address poolToken,
```

```
3       address wethToken,
4       string memory liquidityTokenName,
5       string memory liquidityTokenSymbol
6   )
7       ERC20(liquidityTokenName, liquidityTokenSymbol)
8   {
9   +   assert(wethToken != poolToken);
10      i_wethToken = IERC20(wethToken);
11      i_poolToken = IERC20(poolToken);
12  }
```

[L-4] Event emission has wrong paramter order causing incorrect event emission

Description `TSwapPool::_addLiquidityMintAndTransfer()` emits an event `LiquidityAdded`. Its event signature is `LiquidityAdded(address indexed liquidityProvider, uint256 wethDeposited, uint256 poolTokensDeposited )` but we are passing the parameters in the order `msg.sender, poolTokensToDeposit, wethToDeposit` when it should be `msg.sender, wethToDeposit, poolTokensToDeposit`

Impact Emitted event will have wrong paramter and any system depending on this event will break.

Proof of Concepts Function signature:

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

Usage:

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

Recommended mitigation

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

Informational

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

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

```
4
5     /*//////////////////////////////////////////////////////////////
```

[I-2] `PoolFactory::constructor()` is missing a zero address check for `wethToken`

```
constructor(address wethToken) {
+   assert(wethToken != address(0));
    i_wethToken = wethToken;
}
```

[I-3] `PoolFactory::createPool()` calls `.name()` instead of `.symbol()` for token symbol

```
    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());
    TSwapPool tPool = new TSwapPool(tokenAddress, i_wethToken,
        liquidityTokenName, liquidityTokenSymbol);
    s_pools[tokenAddress] = address(tPool);
```

[I-4] Use `e` notation for denoting multiples of 1000 in `TSwapPool`

line 45:

```
-   uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
+   uint256 private constant MINIMUM_WETH_LIQUIDITY = 1e9;
```

[I-5]: Literal Instead of Constant

Define and use `constant` variables instead of using literals. If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

4 Found Instances

- Found in src/TSwapPool.sol Line: 228

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

- Found in src/TSwapPool.sol Line: 363

```
1e18, i_wethToken.balanceOf(address(this)),
    i_poolToken.balanceOf(address(this))
```

- Found in src/TSwapPool.sol Line: 369

```
1e18, i_poolToken.balanceOf(address(this)),
    i_wethToken.balanceOf(address(this))
```

[I-6] Documentation missing for `TSwapPool::swapExactInput()` and `TSwapPool::swapExactOutput()`

Description Documentation and natspec for `TSwapPool::swapExactInput()` and `TSwapPool::swapExactOutput()` are missing

Recommended mitigation Add documenation and natspec for the two functions

[I-7] Unused variable in `TSwapPool::swapExactInput()`

The `output` variable is never used and can be safely removed

```
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));
```

Gas

[G-1] `TSwapPool::deposit()::poolTokenReserves` is not used anywhere and should be removed to reduce gas fees

```
    uint256 wethReserves = i_wethToken.balanceOf(address(this));
-   uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
    // Our invariant says weth, poolTokens, and liquidity tokens must
        always have the same ratio after the
```

```
4     // initial deposit
```

[G-2] Public Function Not Used Internally

If a function is marked public but is not used internally, consider marking it as `external`.

1 Found Instances

- Found in src/TSwapPool.sol Line: 249

```
1     function swapExactInput(
```

- Found in src/TSwapPool.sol Line: 435

```
1     function totalLiquidityTokenSupply() public view returns (
          uint256) {
```