TSwap Initial Audit Report

Version 1.0

Special for my GitHub

November 6, 2025

TSwap Audit Report

Putfor

November 6, 2025

Prepared by: Putfor https://discordapp.com/users/709350502630162472

Table of Contents

- Medium
  - [M-1] The `TSwapPool::deposit` function ignores `deadline` parameter allowing execution of stale transactions
- Low
  - [L-1] `TSwapPool::LiquidityAdded` event parameters are emitted in incorrect order
  - [L-2] `TSwapPool::swapExactInput` function declares return value but doesn't return it
- Informational
  - [I-1] `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` don't be used and should be remove
  - [I-2] `PoolFactory::constructor(address wethToken)` don't have zero address check
  - [I-3] `PoolFactory::createPool` call `IERC20::name` for `PoolFactory::liquidityTokenSymbol` and should call `IERC20::symbol`
  - [I-4] `TSwapPool::constructor` don't have zero address check for `wethToken` and `poolToken`
  - [I-5] `TSwapPool::poolTokenReserves` is unused variable
  - [I-6] `TSwapPool::deposit` function violates Checks-Effects-Interactions (CEI) pattern
  - [I-7] The `TSwapPool::getOutputAmountBasedOnInput` function have magic numbers in swap calculation reduce code readability and maintainability
  - [I-8] Missing NatSpec documentation for `TSwapPool::swapExactInput` function reduces code clarity
  - [I-9] The `TSwapPool::swapExactInput` function is declared as **`public`** but is only called externally and it should be `external`

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.

Disclaimer

Putfor 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 research by Putfor 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

I use the CodeHawks `https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity` severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

```
2a47715b30cf11ca82e643a8d4c2c802490976b5
38dd009b351b1c8ddadb148704e67652ad679cd8
```

Note: *The following two lines represent a single commit hash that has been split for formatting purposes due to its length exceeding the line width.*

Scope

```
src/
#-- PoolFactory.sol
#-- TSwapPool.sol
```

- Solc Version: 0.8.20
- Chain(s) to deploy contract to: Ethereum
- Tokens:
  - Any ERC20 token

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

This security audit of the TSwapPool smart contract was conducted using a comprehensive methodology combining multiple analysis techniques. The assessment employed static analysis tools including Slither and Aderyn for automated vulnerability detection, supplemented by extensive manual code review to identify logical flaws and business logic vulnerabilities. Additionally, invariant testing was performed using Foundry with Handler-based fuzzing to validate core protocol invariants under various edge cases and unexpected conditions.

The audit revealed several critical issues affecting the protocol's core functionality, including incorrect mathematical calculations leading to substantial financial impacts, missing slippage protection mechanisms, and fundamental logic errors in swap operations. The findings range from high-severity vulnerabilities requiring immediate attention to informational improvements that enhance code quality and maintainability.

All identified issues have been documented with detailed explanations, impact assessments, and practical mitigation recommendations to assist the development team in addressing these concerns effectively.

Issues found

Severity	Number of issues found
High	4
Medium	1

Severity	Number of issues found
Low	2
Info	9
Total	16

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` leads to 10x overcharging

Description: The `getInputAmountBasedOnOutput` function uses an incorrect fee denominator of 10,000 instead of 1,000, resulting in input amount calculations that are approximately 10 times higher than they should be. This affects all swap operations using this function for pricing.

Impact: Users will be charged 10 times the intended input amount for any swap operation that relies on this function, leading to significant financial losses and making the protocol economically non-viable.

Proof of Concept: For a swap where:

`outputAmount` = 100

`inputReserves` = 1,000

`outputReserves` = 1,000

Current incorrect calculation:

```
inputAmount = (1,000 * 100 * 10,000) / ((1,000 - 100) * 997)
            = 1,000,000 / (900 * 997)
            = 1,000,000 / 897,300
            = 1.114
```

Correct calculation:

```
inputAmount = (1,000 * 100 * 1,000) / ((1,000 - 100) * 997)
            = 100,000 / (900 * 997)
            = 100,000 / 897,300
            = 0.1114
```

The current implementation returns ~1.114 tokens instead of the correct ~0.1114 tokens.

Recommended Mitigation: Replace 10,000 with 1,000 in the calculation:

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

[H-2] `TSwapPool::swapExactOutput` lacks slippage protection allowing front-running and unfavorable swaps

Description: The `swapExactOutput` function does not include a maximum input amount parameter (`maxInputAmount`), meaning users cannot specify the maximum amount of input tokens they are willing to spend to receive the exact output amount. This exposes users to potential front-running attacks and unfavorable market conditions.

Impact: Users may receive significantly worse exchange rates than expected, potentially spending excessive amounts of input tokens due to market volatility or manipulation, leading to direct financial losses.

Proof of Concept:

1. User wants exactly 1 WETH and calls `swapExactOutput` with DAI as input
2. Current market rate suggests this should cost ~2000 DAI
3. Front-runner sees the transaction and executes large swaps to manipulate reserves
4. By the time user's transaction executes, the required input amount becomes 3000 DAI
5. User pays 3000 DAI for 1 WETH without any protection

Recommended Mitigation: Add a maxInputAmount parameter and validation:

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    uint256 outputAmount,
    uint256 maxInputAmount,  // Add this parameter
    uint64 deadline
)
    external
    revertIfZero(outputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 inputAmount)
{
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    inputAmount = getInputAmountBasedOnOutput(
```

```
17          outputAmount,
18          inputReserves,
19          outputReserves
20      );
21
22      // Add slippage protection
23      if (inputAmount > maxInputAmount) {
24          revert TSwapPool__InputAmountTooHigh(inputAmount,
                maxInputAmount);
25      }
26
27      _swap(inputToken, inputAmount, outputToken, outputAmount);
28  }
```

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

Description: The `sellPoolTokens` function incorrectly uses `swapExactOutput` instead of `swapExactInput` for its operation. Users specify `poolTokenAmount` as the exact amount of input tokens they want to sell, but the function treats this parameter as the desired output amount when calling `swapExactOutput`, leading to fundamentally incorrect swap logic.

Impact: The function executes swaps with reversed input/output logic, causing users to receive incorrect amounts of WETH and disrupting core protocol functionality. This represents a critical failure in the token selling mechanism.

Proof of Concept:

1. User calls `sellPoolTokens(100)` intending to sell exactly 100 pool tokens
2. Function calls `swapExactOutput(i_poolToken, i_wethToken, 100, ...)`
3. This interprets 100 as the desired WETH output amount, not the pool token input amount
4. The contract calculates how many pool tokens are needed to get 100 WETH
5. User ends up selling a completely different amount of pool tokens than intended
6. If reserves are 1000 pool tokens and 1000 WETH, user would sell ~1126 pool tokens instead of 100

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`)

```
1       function sellPoolTokens(
2           uint256 poolTokenAmount,
```

```
        uint256 minWethToReceive,
        ) external returns (uint256 wethAmount) {
         return swapExactInput(i_poolToken, poolTokenAmount,
            i_wethToken, minWethToReceive, uint64(block.timestamp));
    }
```

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

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

Description: The protocol follows a strict invariant of `x * y = k`. Where: - `x`: The balance of the pool token - `y`: The balance of WETH - `k`: The constant product of the two balances

This means, that whenever the balances change in the protocol, the 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.

Most simply put, the protocol's core invariant is broken.

Proof of Concept: 1. A user swaps 10 times, and collects the extra incentive of `1_000_000_000_000_000_000` tokens 2. That user continues to swap untill all the protocol funds are drained

Place the following into `TSwapPool.t.sol`.

```

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

        uint256 outputWeth = 1e17;

        vm.startPrank(user);
```

```
12          poolToken.approve(address(pool), type(uint256).max);
13          poolToken.mint(user, 100e18);
14          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
15          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
16          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
17          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
18          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
19          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
20          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
21          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
22          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
23
24          int256 startingY = int256(weth.balanceOf(address(pool)));
25          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
26
27          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
28          vm.stopPrank();
29
30          uint256 endingY = weth.balanceOf(address(pool));
31          int256 actualDeltaY = int256(endingY) - int256(startingY);
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

Recommended Mitigation: Remove the extra incentive mechanism. If you want to keep this in, we should account for the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

```
1           // Remove this part of code
2           swap_count++;
3           // Fee-on-transfer
4           if (swap_count >= SWAP_COUNT_MAX) {
5               swap_count = 0;
6               outputToken.safeTransfer(msg.sender, 1
                    _000_000_000_000_000_000);
7           }
```

Medium

[M-1] The `TSwapPool::deposit` function ignores `deadline` parameter allowing execution of stale transactions

Description: The `deposit` function accepts a `uint64 deadline` parameter but fails to validate it against the current `block timestamp`. This allows transactions to be mined even after the user's specified deadline has passed, potentially resulting in unfavorable trade conditions that the user did not intend to accept.

Impact: Users may receive worse exchange rates than expected if their transactions are executed with significant delay, leading to potential financial losses due to price slippage.

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:117:9:
    |
117 |         uint64 deadline
    |         ^^^^^^^^^^^^^^^
```

Low

[L-1] `TSwapPool::LiquidityAdded` event parameters are emitted in incorrect order

Description: The `LiquidityAdded` event is emitted with parameters in the wrong sequence. The event signature expects `(wethDeposited, poolTokensDeposited)` but the emission provides `(poolTokensToDeposit, wethToDeposit)`, swapping the values of WETH and pool token amounts.

Impact: Off-chain monitoring systems, indexers, and user interfaces will display incorrect deposit amounts, showing pool token amount as WETH and vice versa. This causes data inconsistency but doesn't affect contract functionality or funds.

Recommended Mitigation: Correct the parameter order in the emit statement:

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

[L-2] `TSwapPool::swapExactInput` function declares return value but doesn't return it

Description: The `swapExactInput` function is declared to return `uint256 output`, but the function implementation lacks an explicit return statement. While Solidity may implicitly return zero

in such cases, this creates unexpected behavior and violates the function's intended purpose.

Impact: Callers of this function will always receive 0 instead of the actual output amount, breaking integration with other contracts and frontends that rely on the return value to determine the actual tokens received from the swap.

Proof of Concept:

```
Warning (5667): Unused function parameter. Remove or comment out the
    variable name to silence this warning.
   --> src/TSwapPool.sol:308:18:
    |
308 |         returns (uint256 output)
    |
```

Recommended Mitigation: Explicitly return the `outputAmount` variable:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
)
    external
    revertIfZero(inputAmount)
    revertIfDeadlinePassed(deadline)
    returns (uint256 output)
{
    // ... existing calculations ...

    _swap(inputToken, inputAmount, outputToken, outputAmount);

    return outputAmount;
}
```

Informational

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist(address tokenAddress)` don't be used and should be remove

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

[I-2] `PoolFactory::constructor(address wethToken)` don't have zero address check

```
constructor(address wethToken) {
    // for example
    if (wethToken == address(0)) {
        revert();
    }
    i_wethToken = wethToken;
}
```

[I-3] `PoolFactory::createPool` call `IERC20::name` for `PoolFactory::liquidityTokenSymbol` and should call `IERC20::symbol`

```
// wrong
string memory liquidityTokenSymbol = string.concat("ts", IERC20(
    tokenAddress).name());
// correct
string memory liquidityTokenSymbol = string.concat("ts", IERC20(
    tokenAddress).symbol());
```

[I-4] `TSwapPool::constructor` don't have zero address check for `wethToken` and `poolToken`

```
constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    ) ERC20(liquidityTokenName, liquidityTokenSymbol) {

        // for example
        if (wethToken == address(0)) {
            revert();
        }
        if (poolToken == address(0)) {
            revert();
        }

        i_wethToken = IERC20(wethToken);
        i_poolToken = IERC20(poolToken);
    }
```

[I-5] `TSwapPool::poolTokenReserves` is unused variable

```
Warning (2072): Unused local variable.
```

```
   --> src/TSwapPool.sol:131:13:
    |
131 |             uint256 poolTokenReserves = i_poolToken.balanceOf(
    address(this));
    |
```

[I-6] `TSwapPool::deposit` function violates Checks-Effects-Interactions (CEI) pattern

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
            revert TSwapPool__WethDepositAmountTooLow(
                MINIMUM_WETH_LIQUIDITY,
                wethToDeposit
            );
        }
        if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this))
                ;
            uint256 poolTokenReserves = i_poolToken.balanceOf(address(
                this));
            uint256 poolTokensToDeposit =
                getPoolTokensToDepositBasedOnWeth(
                wethToDeposit
            );
            if (maximumPoolTokensToDeposit < poolTokensToDeposit) {
                revert TSwapPool__MaxPoolTokenDepositTooHigh(
                    maximumPoolTokensToDeposit,
                    poolTokensToDeposit
                );
            }

            liquidityTokensToMint =
                (wethToDeposit * totalLiquidityTokenSupply()) /
                wethReserves;
            if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
                revert TSwapPool__MinLiquidityTokensToMintTooLow(
                    minimumLiquidityTokensToMint,
                    liquidityTokensToMint
                );
            }
```

```
            _addLiquidityMintAndTransfer(
                wethToDeposit,
                poolTokensToDeposit,
                liquidityTokensToMint
            );
        } else {
            _addLiquidityMintAndTransfer(
                wethToDeposit,
                maximumPoolTokensToDeposit,
                wethToDeposit
            );
            liquidityTokensToMint = wethToDeposit;
        }
    }
```

[I-7] The `TSwapPool::getOutputAmountBasedOnInput` function have magic numbers in swap calculation reduce code readability and maintainability

Recommended Mitigation: Replace magic numbers with named constants:

```
uint256 constant FEE_NUMERATOR = 997;
uint256 constant PRECISION = 1000;
```

[I-8] Missing NatSpec documentation for `TSwapPool::swapExactInput` function reduces code clarity

Description: The `swapExactInput` function lacks NatSpec documentation, making it difficult for developers and auditors to understand its purpose, parameters, and behavior without analyzing the implementation code.

Recommended Mitigation: Add comprehensive NatSpec documentation.

[I-9] The `TSwapPool::swapExactInput` function is declared as `public` but is only called externally and it should be `external`

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

```
9           revertIfZero(inputAmount)
10          revertIfDeadlinePassed(deadline)
11          returns (uint256 output) {...}
```