T-Swap Protocol Audit Report

Reviewer: Muhammad Faran
Date: June 29, 2025
Repo: Cyfrin/5-t-swap-audit
Commit Hash: `1ec3c30253423eb4199827f59cf564cc575b46db`

Table of Contents

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and view of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation).

Likelihood ↓ / Impact →	High	Medium	Low
High	**High**	**High/Medium**	**Medium**
Medium	**High/Medium**	**Medium**	**Medium-Low**
Low	**Medium**	**Medium-Low**	**Low**

Audit Details

Commit Hash Reviewed: `1ec3c30253423eb4199827f59cf564cc575b46db`
Scope:

- `TSwapPool.sol`
- `PoolFactory.sol`

Roles/Actors

- **Liquidity Providers**: Users who deposit liquidity into the pools and receive LP ERC20 tokens.
- **Users**: Users who swap tokens using the T-Swap protocol.

Protocol Summary

T-Swap is a decentralized Automated Market Maker (AMM) that allows permissionless token swapping without using traditional order books. Liquidity is pooled, and pricing is determined algorithmically.

Executive Summary

Severity	Number Of Issues Found

High	**3**
Medium	**0**
Low	**1**
Informational	**2**

[H-1] Unused Deadline in Deposit Function hence Missed Expiry Check

Scope `TSwapPool.sol`

Description

- Normally, a deadline parameter should enforce a time restriction on a function call, ensuring the transaction is executed within a valid time window.
- However, the `deposit()` function includes a `deadline` parameter but never uses it, leading to users being able to deposit liquidity after their intended expiry.

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    uint64 deadline //@> deadline is not used
)
```

Risk

Likelihood:

- Any user providing liquidity with the expectation of a time constraint will experience unexpected behavior.
- Automated frontends may rely on deadlines, expecting transactions to fail if they are delayed.

Impact:

- May allow unintended deposits beyond the deadline window.
- Can break UI expectations or automated liquidity providers relying on deadline safety.

Proof of Concept

A user signs a deposit transaction with a past `deadline`. The function proceeds normally without any revert, even though the deadline has passed.

```
// Simulated deposit call with expired deadline
pool.deposit(wethAmount, minLPTokens, maxTokens, uint64(block.timestamp - 100));
```

Recommended Mitigation

Enforce the deadline check using the existing modifier.

```diff
- returns (uint256 liquidityTokensToMint)
+ revertIfDeadlinePassed(deadline) returns (uint256 liquidityTokensToMint)
```

[H-2] Incorrect Fee Denominator leads to Inconsistent Invariant Calculation

Scope `TSwapPool.sol`

Description

- Normally, swap math uses a consistent fee structure (e.g., 0.3% fee as 997/1000).
- In `getInputAmountBasedOnOutput`, the function uses `10000` as the denominator instead of `1000`, which is used elsewhere, leading to inconsistent fee application.

```
return ((inputReserves * outputAmount) * 10000) / ((outputReserves - outputAmount) * 997); //@> inconsistent denominator
```

Risk

Likelihood:

- Every time a user swaps for an exact output amount.
- Occurs in all trading scenarios involving reverse swap calculation.

Impact:

- Breaks invariant preservation (`x * y = k` may not hold).
- Results in incorrect input amount calculation and economic imbalances.

Proof of Concept

Calling `swapExactOutput` and comparing with `swapExactInput` will show mismatched fee logic:

```
// swapExactOutput will calculate much higher input requirement than expected
swapExactOutput(...); // uses wrong fee math
```

Recommended Mitigation

Fix the denominator to match the rest of the swap logic.

```diff
- ((inputReserves * outputAmount) * 10000)
+ ((inputReserves * outputAmount) * 1000)
```

[H-3] Incentive Transfer Breaks AMM Invariant

Scope `TSwapPool.sol`

Description

- Normally, Automated Market Makers (AMMs) preserve the constant product invariant `x * y = k`, which ensures that the token pricing remains consistent and tamper-resistant.
- In the `_swap()` function, every 10th swap transfers **1 whole token** (`1e18`) to the swapper without accounting for it in the input/output calculations. This violates the invariant, as it alters the reserves arbitrarily.

```
if (swap_count >= SWAP_COUNT_MAX) {
    swap_count = 0;
    outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000); //@> breaks invariant
}
```

Risk

Likelihood:

- Occurs deterministically every 10th swap.
- Highly likely in active pools with high trading frequency.

Impact:

- Allows economically irrational transfers from the pool, opening arbitrage and draining vectors.
- Causes divergence between on-chain state and AMM pricing model (`x * y = k` no longer holds).

Proof of Concept

You can write a test that confirms users receive extra tokens without providing any additional input:

```
for (uint i = 0; i < 10; i++) {
    swapExactInput(tokenIn, amountIn, tokenOut, minOut, deadline);
    // The 10th swap will return more tokens than the invariant allows.
}
```

Recommended Mitigation

Remove the incentive logic from `_swap()` entirely to protect the invariant. If you want to reward traders, implement a separate, controlled reward mechanism outside the core AMM logic.

```diff
- if (swap_count >= SWAP_COUNT_MAX) {
-     swap_count = 0;
```

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

[L-1] Using Non-Standard IERC20 Interface

Scope `PoolFactory.sol`

Description

- Normally, OpenZeppelin's `IERC20` is used for interface consistency and ABI compatibility.
- This code uses `IERC20` from `forge-std`, which doesn't include `name()` or `symbol()`, leading to runtime or compile-time errors.

```
//@audit import IERC20 from Oppenzeppelin instead
import { IERC20 } from "forge-std/interfaces/IERC20.sol"; //@> wrong import
```

Risk

Likelihood:

- Every time `createPool()` is called.
- Will break unless the token implements optional metadata extensions.

Impact:

- Causes runtime errors if `.name()` or `.symbol()` is not implemented.
- Breaks compatibility with many ERC20s.

Proof of Concept

Call `createPool()` with a basic ERC20 token that lacks `name()`.

```
// Fails due to missing .name()
createPool(basicERC20);
```

Recommended Mitigation

Use OpenZeppelin's `IERC20Metadata` interface which explicitly defines `name()` and `symbol()`.

```diff
- import { IERC20 } from "forge-std/interfaces/IERC20.sol";
+ import { IERC20Metadata as IERC20 } from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
```

[I-1] Unused Variable in Deposit Function leading towards Dead Code

Scope `TSwapPool.sol`

Description

- Normally, variables are declared for use in logic, comparisons, or calculations.
- The `poolTokenReserves` is declared in the `deposit()` function but is never used, which adds unnecessary computation and indicates possible incomplete logic.

```
uint256 wethReserves = i_wethToken.balanceOf(address(this));
uint256 poolTokenReserves = i_poolToken.balanceOf(address(this)); //@> unused variable
```

Risk

Likelihood:

- Happens every time liquidity is added to an existing pool.
- Common in early-stage or modified code that doesn’t complete a prior logic update.

Impact:

- Reduces code clarity and introduces tech debt.
- Could mislead auditors or future developers into thinking it has functional importance.

Proof of Concept

Simply reviewing the function shows the value is fetched but unused, and removing it has no impact.

```
// Commenting out this line has no effect on behavior
// uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

Recommended Mitigation

Remove the dead variable to improve readability.

```diff
- uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
+ // Removed unused variable
```

[I-2] Unused Return in `swapExactInput` hence Missing Assignment

Scope `TSwapPool.sol`

Description

- Normally, if a function declares a return value, that value should be assigned and returned.

- `swapExactInput()` declares a `returns (uint256 output)` but never assigns a value to `output`, resulting in always returning 0.

```
returns (uint256 output) //@> output is not used
```

Risk

Likelihood:

- Happens on every call to `swapExactInput`.
- Affects any integration relying on the return value.

Impact:

- Users and integrators will always receive `0` as output, even if the swap succeeds.
- Misleads developers and breaks client contract expectations.

Proof of Concept

Calling the function returns zero despite a successful swap.

```
uint256 out = swapExactInput(...); // out == 0 always
```

Recommended Mitigation

- Return the actual output value.

```diff
+ return outputAmount;
```
