T-Swap Audit Report

Version 1.0

0xPexy

June 28, 2025

T-Swap Audit Report

0xPexy

Jun 28, 2025

Prepared by: 0xPexy

Table of Contents

Protocol Summary

T-Swap 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.

Disclaimer

0xPexy 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 0xPexy 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

I use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Commit Hash

`1ec3c30253423eb4199827f59cf564cc575b46db`

Scope

```
- 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	2
Low	2
Info	5
Total	13

Findings

High Severity

[H-1] Bonus Payouts in _swap Break Core Invariant, Leading to Pool Drain

Description: In the `TSwapPool::_swap` function, there is an extra incentive per 10 swaps, transfers 1e18 bonus output tokens to the address.

Impact: This breaks the *core invariant* that x*y=k because it removes 1e18 of the outputToken from the pool without a corresponding input. This systematically drains value from the pool with each bonus payout, causing a direct loss of funds for liquidity providers.

Proof of Concept: The code shows that the pool consist of 1:1 PoolToken-WETH with 10000(e18) amount each. The swapper swaps 10 times then the invariant is broken.

Code

1. Add the followings into `test/unit/InvariantTest.t.sol`.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import {Test} from "forge-std/Test.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {TSwapPool} from "../../src/TSwapPool.sol";

contract InvariantTest is Test {
    ERC20Mock pt;
    ERC20Mock weth;
    TSwapPool pool;

    uint256 constant INIT_PT = 10000e18;
    uint256 constant INIT_WETH = 10000e18;
    address lp = makeAddr("lp");
    address swapper = makeAddr("swapper");
    uint256 constant INIT_BAL = type(uint128).max;

    int256 public expectedDeltaWETH;
    int256 public actualDeltaWETH;

    modifier useSwapper() {
        vm.startPrank(swapper);
        _;
        vm.stopPrank();
    }

    function setUp() public {
        pt = new ERC20Mock();
        weth = new ERC20Mock();
        pool = new TSwapPool(address(pt), address(weth), "LP", "LP");

        pt.mint(lp, INIT_PT);
        weth.mint(lp, INIT_WETH);
        pt.mint(swapper, INIT_PT);
        weth.mint(swapper, INIT_WETH);

        vm.startPrank(lp);
        pt.approve(address(pool), UINT256_MAX);
        weth.approve(address(pool), UINT256_MAX);

        pool.deposit(INIT_WETH, INIT_WETH, INIT_PT, uint64(block.timestamp));

        vm.stopPrank();
    }

    // hook for testing
    function getPoolReserves() public view returns (int256, int256) {
        return (
            int256(pt.balanceOf(address(pool))),
            int256(weth.balanceOf(address(pool)))
        );
    }

    function testInvariantBreak() public {
        uint loops = 10;
        for (uint i = 0; i < loops; ++i) {
            swapByWETH();
            assertEq(expectedDeltaWETH, actualDeltaWETH);
        }
    }

    // swap PT->WETH by WETH amount
    function swapByWETH() public useSwapper {
        // 1. bound input
        uint256 amountWETH = 1e18 + 12345;
        int256 beforePT;
        int256 beforeWETH;
        int256 afterPT;
        int256 afterWETH;

        (beforePT, beforeWETH) = getPoolReserves();
        uint256 amountPT = pool.getInputAmountBasedOnOutput(
            amountWETH,
            uint256(beforePT),
            uint256(beforeWETH)
        );

        // 2. set invariants
        expectedDeltaWETH = (-1) * int256(amountWETH);

        // 3. run pre-cond. tx
        pt.approve(address(pool), amountPT);

        // 4. run tx
        pool.swapExactOutput(pt, weth, amountWETH, uint64(block.timestamp));

        // 5. update ghost vars
        (afterPT, afterWETH) = getPoolReserves();
        actualDeltaWETH = afterWETH - beforeWETH;
    }
}
```

2. Running `forge test --mt testInvariantBreak -vv`, the assertion fails with the difference 1e18, which is hard-coded in the `_swap`. This means the pool has less balance because the extra rewards transferred to the swapper.

```
[FAIL: assertion failed: -1000000000000012345 != -2000000000000012345] testInvariantBreak()
```

Recommended Mitigation: Remove the extra reward.

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

[H-2] Incorrect Fee Calculation in `getInputAmountBasedOnOutput` Overcharges Users

Description: `TSwapPool::getInputAmountBasedOnOutput` calculates `inputAmount` by multiplying `10_000` to `(inputReserves * outputAmount)`.

Impact: Because the `TSwapPool::swapExactOutput` uses the method, causes users to pay ten times more inputs than the normal case to get the same amount of outputs.

Proof of Concept: The code shows that the pool consist of 1:1 PoolToken-WETH with 100(e18) amount each. The user wants 10 output WETH, expecting about 11.11 PoolTokens are inserted to the pool. But about 111 PoolTokens are inserted, taken from the user.

Code

1. Add the followings into the `test/unit/TSwapPool.t.sol`.

```
    function testIncorrctInputAmount() 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);
        // mint & approve sufficient pool token
        poolToken.mint(user, 1000e18);
        poolToken.approve(address(pool), 1000e18);

        // swap poolToken -> 10 weth
        // in pool, there should be about 111.11 poolToken
        // considering 0.03% fee, bound as 112
        uint256 expectedMaxPoolBalance = 112e18;
        uint256 output = 10e18;

        pool.swapExactOutput(poolToken, weth, output, uint64(block.timestamp));
        assertGe(expectedMaxPoolBalance, poolToken.balanceOf(address(pool)));
    }
```

2. Run `forge test --mt testIncorrctInputAmount -vv` to see the result below.

```
[FAIL: assertion failed: 112000000000000000000 < 211445447453471525688] testIncorrctInputAmount() (gas: 283472)
```

Recommended Mitigation: Correct the numerator. This passes the test above.

```
- return ((inputReserves * outputAmount) * 10000) / ((outputReserves - outputAmount) * 997);
+ return ((inputReserves * outputAmount) * 1_000) / ((outputReserves - outputAmount) * 997);
```

[H-3] `sellPoolTokens` Uses Incorrect Swap Logic, Causing Users to Sell Wrong Amount

Description: `TSwapPool::sellPoolTokens` is intended to facilitate users selling pool tokens in exchange of WETH, calls `swapExactOutput` with `poolTokenAmount` parameter. This function fixes the expected WETH amount to `poolTokenAmount` and calculate the amount of pool tokens to sell internally.

Impact: Users may think that they sell expected amount of pool tokens, but wrong amount is calculated and get an unexpected swap result.

Proof of Concept: The code shows that the pool consist of 1:1 PoolToken-WETH with 100(e18) amount each. The user sells 10 PoolTokens but about 10 times larger PTs are sold.

Code

1. Add the followings into the `test/unit/TSwapPool.t.sol`.

```
    function testIncorrectSellPoolTokens() 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);
        // mint & approve sufficient pool token
        poolToken.mint(user, 1000e18);
        poolToken.approve(address(pool), 1000e18);

        // swap 10 poolToken -> ~= 9.1 WETH
        uint256 userPTBalance = poolToken.balanceOf(user);
        uint256 expectedInput = 10e18;

        pool.sellPoolTokens(expectedInput);
        assertEq(expectedInput, userPTBalance - poolToken.balanceOf(user));
    }
```

2. Running `forge test --mt testIncorrectSellPoolTokens -vv` shows the output below.

```
[FAIL: assertion failed: 10000000000000000000 != 11144544745347152568]
    testIncorrectSellPoolTokens() (gas: 283733)
```

Recommended Mitigation: Use `swapExactInput` instead of `swapExactOutput`.

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

Apply the followings into the `testIncorrectSellPoolTokens` to pass the test.

```
        uint256 userPTBalance = poolToken.balanceOf(user);
        uint256 expectedInput = 10e18;
+       uint256 expectedMinOutput = 9e18;

-       pool.sellPoolTokens(expectedInput);
+       uint256 acutalOutput = pool.sellPoolTokens(expectedInput, expectedMinOutput);

        assertEq(expectedInput, userPTBalance - poolToken.balanceOf(
            user));
+       assertLe(expectedMinOutput, acutalOutput);
```

[H-4] `swapExactOutput` Misses Bounding Input Amount, Causing Excessive Slippages

Description: The `TSwapPool::swapExactOutput` misses amount limitation for the input token compared to `swapExactInput` checks the minimum output token amount to receive.

Impact: Users might overpay the input token for buying the output token than they willing to pay.

Proof of Concept: The code shows that the pool consist of 100e18 PoolTokens and 10e18 WETH. The user want to get 1 WETH and expected to transfer about 11.4 PT to pool. But the attacker formally takes 5 WETH from the pool, user spends about 276.6 PT to buy one WETH.

Code

1. Add the followings into the `test/unit/TSwapPool.t.sol`.

```
    function testMissingSlippageProtection() public {
        uint256 INIT_POOL_PT = 100e18;
        uint256 INIT_POOL_WETH = 10e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), INIT_POOL_WETH);
        poolToken.approve(address(pool), INIT_POOL_PT);
        pool.deposit(
            INIT_POOL_WETH,
            INIT_POOL_WETH,
            INIT_POOL_PT,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        uint256 INIT_AMOUNT = 10000e18;
        address attacker = makeAddr("attacker");
        poolToken.mint(user, INIT_AMOUNT);
        poolToken.mint(attacker, INIT_AMOUNT);

        uint256 userOutput = 1e18;
        uint256 expectedUserInput = pool.getInputAmountBasedOnOutput(
            userOutput,
            INIT_POOL_PT,
            INIT_POOL_WETH
        );

        uint256 attackerOutput = 5e18;
        vm.startPrank(attacker);
        poolToken.approve(address(pool), INIT_AMOUNT);
        pool.swapExactOutput(
            poolToken,
            weth,
            attackerOutput,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        vm.startPrank(user);
        poolToken.approve(address(pool), INIT_AMOUNT);
        uint256 actualUserInput = pool.swapExactOutput(
            poolToken,
            weth,
            userOutput,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        assertEq(expectedUserInput, actualUserInput);
    }
```

2. Running `forge test --mt testMissingSlippageProtection -vv` shows the output below.

```
[FAIL: assertion failed: 111445447453471525688 != 2765820027786468734185] testMissingSlippageProtection() (gas: 383992)
```

Recommended Mitigation: Consider applying the followings.

```
+   error TSwapPool__InputTooHigh(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__InputTooHigh(inputAmount, maxInputAmount);
+       }

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

Medium Severity

[M-1] Missing Deadline Check in `deposit` Allows Transactions After Deadline

Description: `TSwapPool::deposit` has `deadline` parameter, intended to reject transactions after the deadline. However, `deadline` is unused anywhere, results to missing a deadline check.

Impact: Users willing to deposit in specific period considering the market conditions may submit the transaction with the deadline. But this will not be blocked and exectued in a worse price than they intended.

Proof of Concept: Run `make build` to see a compilation warning.

```
Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> src/TSwapPool.sol:105:9:
    |
105 |         uint64 deadline
    |         ^^^^^^^^^^^^^^^
```

Recommended Mitigation: Add a deadline check in `deposit`.

```
function deposit(...) external
    revertIfZero(wethToDeposit)
+   revertIfDeadlinePassed(deadline)
```

[M-2] Protocol Fails to Account for Rebase, Fee-on-Transfer and ERC-777 Tokens, Breaking the Core Invariant

Description: The *Weird-ERC20* tokens like rebase, fee-on-transfer and ERC-777 have abnormal transfers. If a pool includes these tokens, the sum of the user and the pool balance can be changed during a swap.

Impact: These tokens might break the core invariant x*y=k in the pool, because the x or y can be changed.

Proof of Concept: The code shows that the pool consist of 1:1 PoolToken-WETH with 10000(e18) amount each. PoolToken is a fee-on-transfer token which sends 10% of transferring amount to the owner. The swapper swaps 1 PoolToken to WETH, doing 10 times then the invariant is broken.

Code

1. Add the followings into `test/unit/WeirdERC20PoolTest.t.sol`.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

import {Test} from "forge-std/Test.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";
import {WeirdERC20} from "../mocks/WeirdERC20.sol";

import {TSwapPool} from "../../src/TSwapPool.sol";

contract WeirdERC20PoolTest is Test {
    WeirdERC20 pt;
    ERC20Mock weth;
    TSwapPool pool;

    uint256 constant INIT_PT = 10000e18;
    uint256 constant INIT_WETH = 10000e18;
    address lp = makeAddr("lp");
    address swapper = makeAddr("swapper");
    address weirdERC20Owner = makeAddr("weirdERC20Owner");
    uint256 constant INIT_BAL = type(uint128).max;

    int256 public expectedDeltaPT;
    int256 public actualDeltaPT;

    modifier useSwapper() {
        vm.startPrank(swapper);
        _;
        vm.stopPrank();
    }

    function setUp() public {
        vm.prank(weirdERC20Owner);
        pt = new WeirdERC20();

        weth = new ERC20Mock();
        pool = new TSwapPool(address(pt), address(weth), "LP", "LP");

        pt.mint(lp, INIT_PT);
        weth.mint(lp, INIT_WETH);
        pt.mint(swapper, INIT_PT);
        weth.mint(swapper, INIT_WETH);

        vm.startPrank(lp);
        pt.approve(address(pool), UINT256_MAX);
        weth.approve(address(pool), UINT256_MAX);

        pool.deposit(INIT_WETH, INIT_WETH, INIT_PT, uint64(block.timestamp));

        vm.stopPrank();
    }

    // hook for testing
    function getPoolReserves() public view returns (int256, int256) {
        return (
            int256(pt.balanceOf(address(pool))),
            int256(weth.balanceOf(address(pool)))
        );
    }

    function testWeirdERC20() public {
        uint loops = 10;
        for (uint i = 0; i < loops; ++i) {
            swapByPT();
            assertEq(expectedDeltaPT, actualDeltaPT);
        }
    }

    // swap PT->WETH by WETH amount
    function swapByPT() public useSwapper {
        // 1. bound input
        uint256 amountPT = 1e18;
        int256 beforePT;
        int256 beforeWETH;
        int256 afterPT;
        int256 afterWETH;

        (beforePT, beforeWETH) = getPoolReserves();
        uint256 amountWETH = pool.getOutputAmountBasedOnInput(
            amountPT,
            uint256(beforePT),
            uint256(beforeWETH)
        );

        // 2. set invariants
        expectedDeltaPT = int256(amountPT);

        // 3. run pre-cond. tx
        pt.approve(address(pool), amountPT);

        // 4. run tx
        pool.swapExactInput(
            pt,
            amountPT,
            weth,
            amountWETH,
            uint64(block.timestamp)
        );

        // 5. update ghost vars
        (afterPT, afterWETH) = getPoolReserves();
        actualDeltaPT = afterPT - beforePT;
    }
}
```

2. Running `forge test --mt WeirdERC20PoolTest -vv`, the assertion fails with the difference 1e17, meaning 10% of 1e18 PoolToken amount has gone.

```
[FAIL: assertion failed: 1000000000000000000 != 900000000000000000] testWeirdERC20() (gas: 617562)
```

3. If you run the test with `-vvvv`, you can see that 1e17 amount has been transferred to the owner.

```
emit Transfer(from: swapper: [0x4A9D6b0b19CBFfCB0255550661eCB7014283c60E], to: weirdERC20Owner: [0xE8C723E79F10df14c40c3c342395DA8Bbe257f18], value: 100000000000000000 [1e17])
```

Recommended Mitigation: Add the core invariant checks in swap and deposit to track the K always grows.

```
+   // tracks core invariant x*y=k
+   uint256 K;

+   // add in swap, deposit
+      (uint256 ptBalance, uint256 wethBalance) = _getReserves();
+      uint256 newK = ptBalance * wethBalance;
+      // K must grows
+      require(newK >= K);
+      K = newK;

+   // optional hooks
+  function _getReserves() internal view returns (uint256, uint256) {
+      return (
+          i_poolToken.balanceOf(address(this)),
+          i_wethToken.balanceOf(address(this))
+      );
+  }
```

Low Severity

[L-1] Incorrect Parameter Order in Event Might Cause Potential Bugs in Subscribers

Description: There is an incorrect parameter ordering in `TSwapPool::_addLiquidityMintAndTransfer`, which might cause potential bugs in off-chain Apps subscribing the event.

```
contract TSwapPool is ERC20 {
    event LiquidityAdded(address indexed liquidityProvider, uint256 wethDeposited, uint256 poolTokensDeposited);
    ...
    function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
    )
        private
    {
        ...
        emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
        ...
    }
}
```

Recommended Mitigation: Correct the parameter order.

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

[L-2] Missing Return Value in `swapExactInput` Might Cause Potential Bugs in Other Contracts

Description: `TSwapPool::swapExactInput` has return value `uint256 output`, but never return any value. This might cause potential bugs in the other contracts interacting with the function.

Recommended Mitigation: Return the exact value.

```
function swapExactInput(...) returns (uint256 output) {
    ...
-   uint256 outputAmount = getOutputAmountBasedOnInput(...);
+   output = getOutputAmountBasedOnInput(...);
    if (output < minOutputAmount) {
        revert ...
    }
-   _swap(inputToken, inputAmount, outputToken, outputAmount);
+   _swap(inputToken, inputAmount, outputToken, output);
}
```

Informational

[I-1] Unused Statements

Remove unused statements.

- `error PoolFactory__PoolDoesNotExist(address tokenAddress);` in `PoolFactory`
- `uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));` in `TSwapPool::deposit`

[I-2] Lacking Zero-address Checks

Add zero-address checks in below parts.

- `PoolFactory::constructor`: `address wethToken`

- `TSwapPool::constructor`: `address poolToken` and `address wethToken`

```
// PoolFactory.t.sol
constructor(address wethToken) {
+   require(wethToken != address(0));
    i_wethToken = wethToken;
}

// TSwapPool.t.sol
constructor(
    address poolToken,
    address wethToken,
    ...
)
{
+    require(wethToken != address(0));
+    require(poolToken != address(0));
    i_wethToken = IERC20(wethToken);
    i_poolToken = IERC20(poolToken);
}
```

[I-3] `createPool` Should Use .symbol() for LP Token Symbol

In `PoolFactory::createPool`, consider using `IERC20::symbol` to represent LP token symbol. The `IERC20::name` is already used.

```
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());
```

[I-4] Unnecessary Visibility

The **`public`** function `TSwapPool::swapExactInput` is not internally referenced, use `external`.

[I-5] Unnamed Numeric Constants

Use named numeric constants for arithmetic operations.

```
+   uint256 private constant WITHOUT_FEE = 997;
+   uint256 private constant SCALE = 1000;
```

```
-    uint256 inputAmountMinusFee = inputAmount * 997;
-    uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
+    uint256 inputAmountMinusFee = inputAmount * WITHOUT_FEE;
+    uint256 denominator = (inputReserves * SCALE) + inputAmountMinusFee;
-    return ((inputReserves * outputAmount) * 10000) / ((outputReserves - outputAmount) * 997);
+    return ((inputReserves * outputAmount) * SCALE) / ((outputReserves - outputAmount) * WITHOUT_FEE);
```