Protocol Audit Report

Version 1.0

io10

January 5, 2025

Protocol Audit Report

io10

04/01/2025

Prepared by: [io10] Lead Auditors:

- io10

Table of Contents

T-Swap 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. To understand Uniswap, please watch this video: Uniswap Explained

Disclaimer

io10 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 me 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
- In Scope:

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

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

Actors / 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

The TSwap protocol audit revealed critical vulnerabilities that compromise its core functionality and user trust. Most notably, issues with protocol invariants, fee calculations, and missing safeguards could lead to system instability, overcharging of users, and susceptibility to manipulation.

The audit highlighted that certain design choices, such as added user incentives, break the protocol's fundamental mathematical assumptions, potentially destabilizing the liquidity pool. Additionally, inconsistencies in fee structures and a lack of slippage protection expose users to unexpected costs and unfair trading conditions. Missing deadline enforcement in key functions also risks denial-of-service scenarios and user frustration.

Addressing these issues is paramount to maintaining the protocol's integrity and ensuring predictable, fair outcomes for users. Key recommendations include revising core logic to uphold protocol invariants, ensuring consistency in fee calculations, and implementing safeguards against slippage and missed deadlines.

By resolving these vulnerabilities, TSwap can strengthen its foundation, safeguard user trust, and continue fostering a reliable decentralized trading experience.

Issues found

Severity	Number Of Issues Found
High	5
Medium	0
Low	1
Informational	0
Total	6

Findings

High

[H-1] Protocol Core Invariant Broken which breaks down the core tswap system

Description: The core invariant of the protocol is detailed in the documentation of the tswap protocol and it is broken due to added functionality to reward users for using tswap. See below:

Code

```
/**
     * @notice Swaps a given amount of input for a given amount of
       output tokens.
     * @dev Every 10 swaps, we give the caller an extra token as an
       extra incentive to keep trading on T-Swap.
     * @param inputToken ERC20 token to pull from caller
     * @param inputAmount Amount of tokens to pull from caller
     * @param outputToken ERC20 token to send to caller
     * @param outputAmount Amount of tokens to send to caller
     */
    function _swap(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 outputAmount
    ) private {
        if (
            _isUnknown(inputToken) ||
            _isUnknown(outputToken) ||
            inputToken == outputToken
        ) {
            revert TSwapPool__InvalidToken();
        }

        swap_count++;
        if (swap_count >= SWAP_COUNT_MAX) {
            swap_count = 0;
            outputToken.safeTransfer(msg.sender, 1
                _000_000_000_000_000_000);
        }
        emit Swap(
            msg.sender,
            inputToken,
            inputAmount,
            outputToken,
            outputAmount
        );

        inputToken.safeTransferFrom(msg.sender, address(this),
           inputAmount);
        outputToken.safeTransfer(msg.sender, outputAmount);
    }
```

in the natspec, you will see that it says Every 10 swaps, we give the caller an extra token as an extra incentive to keep trading on T-Swap. This is what breaks the assertion. In the code, you can see a swap_count variable being incremented and once it goes past a certain threshold, an extra amount is sent to msg.sender. This token incentive is what breaks the invariant of this protocol.

Impact: The tswap system fails due to the protocol invariant breaking

Proof of Concept: See the Tswapinvarianttests.t.sol file for the invariant tests and Handler.sol file for the invariant test that exposes the assertion break. Run `forge test --mt invariant_coreinvariantTswap` -vvvv to see results. If fuzz results are hard to read , here is the unit test version that mimics the fuzzer calls and the invariant break.

Code

```
function test_10swapsbreaksinvariant() 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);
        uint256 expected = 9e18;
        poolToken.approve(address(pool), 300e18);
        for (uint256 i = 0; i < 9; i++) {
            pool.swapExactInput(
                poolToken,
                5e18,
                weth,
                1e18,
                uint64(block.timestamp)
            );
        }
        uint256 preoutputreserves = weth.balanceOf(address(pool));
        uint256 expectedoutputAmount = pool.getOutputAmountBasedOnInput
           (
            20e18,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );
        pool.swapExactInput(
            poolToken,
            20e18,
            weth,
            7e18,
            uint64(block.timestamp)
        );
        uint256 postoutputreserves = weth.balanceOf(address(pool));
        uint256 actualoutputAmount = preoutputreserves -
            postoutputreserves;
        console.log(expectedoutputAmount, actualoutputAmount);
        assert(actualoutputAmount > expectedoutputAmount);
        vm.stopPrank();
    }
```

In the above test, a user performs 10 swaps. Before the 10th swap, the expected output amount is measured as well as the weth balance of the pool. After the 10th swap, the weth balance of the pool is measured and these values are compared to the expected amount of weth to leave the pool and you will see that the actualoutputamount is 1 weth more than expected which accounts for the 1eth incentive given to a user after every 10 swaps which breaks the protocol invariant.

Recommended Mitigation: Remove token incentive from `TSwapPool::_swap` function to maintain protocol core invariant.

[H-2] No deadline checks in the `TSwapPool::deposit` function which causes a DOS for the user calling deposit function who sets a deadline

Description: See deposit function:

Code

```
// @notice Adds liquidity to the pool
    /// @dev The invariant of this function is that the ratio of WETH,
        PoolTokens, and LiquidityTokens is the same
    /// before and after the transaction
    /// @param wethToDeposit Amount of WETH the user is going to
        deposit
    /// @param minimumLiquidityTokensToMint We derive the amount of
        liquidity tokens to mint from the amount of WETH the
    /// user is going to deposit, but set a minimum so they know approx
         what they will accept //q if the minimum deviates a lot from
        the actual amount of liquidity tokens minted, how does this give
         the user any idea what they will accept ?

    /// @param maximumPoolTokensToDeposit The maximum amount of pool
        tokens the user is willing to deposit, again it's
    /// derived from the amount of WETH the user is going to deposit //
        q how is this derived if the user has to enter a random amount ?
    /// @param deadline The deadline for the transaction to be
        completed by
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline //bug no deadline checks here with
           revertIfDeadlinePassed(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
                );
            }

            // We do the same thing for liquidity tokens. Similar math.
            liquidityTokensToMint =
                (wethToDeposit * totalLiquidityTokenSupply()) /
                wethReserves;
            if (liquidityTokensToMint < minimumLiquidityTokensToMint) {
                revert TSwapPool__MinLiquidityTokensToMintTooLow(
                    minimumLiquidityTokensToMint,
                    liquidityTokensToMint
                );
            }
            _addLiquidityMintAndTransfer(
                wethToDeposit,
                poolTokensToDeposit,
                liquidityTokensToMint
            );
        } else {
            // This will be the "initial" funding of the protocol. We
                are starting from blank here!
            // We just have them send the tokens in, and we mint
                liquidity tokens based on the weth
            _addLiquidityMintAndTransfer(
                wethToDeposit,
                maximumPoolTokensToDeposit,
                wethToDeposit
            );
            liquidityTokensToMint = wethToDeposit;
        }
    }
```

There are no deadline checks in the deposit function which means that if a user sets a deadline for a deposit and expects it to fail past a certain block , the deposit will be processed leading to a DOS.

Impact: Unexpected behaviour from user calling deadline function

Proof of Concept:

Code

```
function test_depositwdeadline() public {
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        vm.warp(5000);
        vm.roll(5);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp -
            300));

        assertEq(pool.balanceOf(liquidityProvider), 100e18);
        assertEq(weth.balanceOf(liquidityProvider), 100e18);
        assertEq(poolToken.balanceOf(liquidityProvider), 100e18);

        assertEq(weth.balanceOf(address(pool)), 100e18);
        assertEq(poolToken.balanceOf(address(pool)), 100e18);
    }
```

Recommended Mitigation: Include the `TSwapPool:: revertIfDeadlinePassed(deadline)` in the deposit function which includes deadline logic.

[H-3] Fees are inconsistent between `TSwapPool:: swapExactInput` and `TSwapPool:: swapExactOutput` leading to swappers paying more in fees when using `TSwapPool:: swapExactOutput`

Description: The logic for `TSwapPool:: swapExactInput` was based on the following formula which baked the 0.3% fee into each swap:

```
 uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * 1000) +
            inputAmountMinusFee;
        return numerator / denominator;
```

For `TSwapPool:: swapExactOutput`, the logic was:

```
(inputReserves * outputAmount * 10000)((outputReserves - outputAmount)
    * 997);
```

As you can see, the fee calculation is skewed due to the extra 0 in the above formula.

Impact: As a result of the above description, users who choose to swap using `TSwapPool::swapExactOutput` end up paying a 3% fee instead of the afrementioned 0.3% intended by the protocol.

Proof of Concept:

Code

```
    function testCollectFeesInput() 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);
          uint256 expected = 9e18;
          poolToken.approve(address(pool), 10e18);
          pool.swapExactInput(
              poolToken,
              10e18,
              weth,
              expected,
              uint64(block.timestamp)
          );
          vm.stopPrank();

          vm.startPrank(liquidityProvider);
          pool.approve(address(pool), 100e18);
          pool.withdraw(100e18, 90e18, 100e18, uint64(block.timestamp));
          console.log(
              weth.balanceOf(liquidityProvider) +
                  poolToken.balanceOf(liquidityProvider)
          );
          assertEq(pool.totalSupply(), 0);
          assert(
              weth.balanceOf(liquidityProvider) +
                  poolToken.balanceOf(liquidityProvider) >
                  200e18
          );
      }

      function testCollectFeesOutput() 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);
        uint256 expected = 9e18;
        poolToken.approve(address(pool), 10e18);
        weth.approve(address(pool), 100e18);
        pool.swapExactOutput(
            weth,
            poolToken,
            expected,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        vm.startPrank(liquidityProvider);
        pool.approve(address(pool), 100e18);
        pool.withdraw(100e18, 90e18, 90e18, uint64(block.timestamp));
        console.log(
            weth.balanceOf(liquidityProvider) +
                poolToken.balanceOf(liquidityProvider)
        );
        assertEq(pool.totalSupply(), 0);
        assert(
            weth.balanceOf(liquidityProvider) +
                poolToken.balanceOf(liquidityProvider) >
                200e18
        );
    }
```

Inspecting the logs from both of these tests will highlight the fee discrepancies when calling `TSwapPool:: swapExactInput` vs `TSwapPool:: swapExactOutput`

Recommended Mitigation: Refactor the `Tswap::getInputAmountBasedOnOutput` function which is used in the `TSwapPool:: swapExactOutput` to calculate input amount. See below:

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

[H-4] Lack of slippage protection in `TSwapPool:: swapExactOutput` which allows for MEV and causes users to pay unexpected input amounts for swaps

Description: In the `TSwapPool:: swapExactOutput` function, notice how there is no maxInputAmount parameter. This is a problem because if a big transaction frontruns the users transaction, it will cause a slip in the inputAmount calculation which can lead to the user paying more than they wanted to in order to swap their tokens which can be an issue if the slippage is high.

Impact: User paying unexpected and excessive inputAmounts for swaps

Proof of Concept:

Code

```
function test_unexpectedslippage() 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 expected = 9e18;
        uint256 inputreserves = weth.balanceOf(address(pool));
        uint256 outputreserves = poolToken.balanceOf(address(pool));
        uint256 expecteduserinput = pool.getInputAmountBasedOnOutput(
            expected,
            inputreserves,
            outputreserves
        );
        console.log(expecteduserinput);

        vm.startPrank(user2);
        weth.approve(address(pool), 100e18);
        pool.swapExactOutput(
            weth,
            poolToken,
            expected,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        uint256 newinputreserves = weth.balanceOf(address(pool));
        uint256 newoutputreserves = poolToken.balanceOf(address(pool));
        uint256 actualuserinput = pool.getInputAmountBasedOnOutput(
            expected,
            newinputreserves,
            newoutputreserves
        );

        vm.startPrank(user);
        weth.approve(address(pool), 300e18);
        pool.swapExactOutput(
            weth,
            poolToken,
            expected,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        console.log(actualuserinput, expecteduserinput);
```

```
            assert(actualuserinput > expecteduserinput);
        }
```

Recommended Mitigation: Include a maxInputAmount parameter in `TSwapPool::swapExactOutput` function that will allow the user to set a maximum input amount they are willing to pay for a specified output and if inputAmount goes above this, then tx should revert. See modified function below:

Code

```

function swapExactOutput(
            IERC20 inputToken,
            IERC20 outputToken,
+           uint256 maxInputAmount,
            uint256 outputAmount,
            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);
        }
```

[H-5] Wrong Logic in `TswapPool::sellPoolTokens` which forces users to sell more pool tokens than their intended amount

Description: In `TswapPool::sellPoolTokens` , see the description of the function below:

Code

```

/**
```

```
     * @notice wrapper function to facilitate users selling pool tokens
         in exchange of WETH
     * @param poolTokenAmount amount of pool tokens to sell
     * @return wethAmount amount of WETH received by caller
     */
    function sellPoolTokens(
        uint256 poolTokenAmount
    ) external returns (uint256 wethAmount) {
        return
            swapExactOutput(
                i_poolToken,
                i_wethToken,
                poolTokenAmount,
                uint64(block.timestamp)
            );
    }
```

`TswapPool::pooltokenamount` is a parameter where the user should enter the amount of pool tokens they want to sell. By calling `TswapPool::swapExactOutput` in this function, `TswapPool::pooltokenamount` is set as the output amount. i.e. the amount of weth that the user wants to receive. As a result, the function ends up taking a larger input amount than `TswapPool::pooltokenamount` which is not expected behaviour and in some cases, can lead to users swapping a larger amount of pool tokens than they intended

Impact: Users are forced to sell more pool tokens than their intended amount

Proof of Concept:

Code

```

function test_sellpooltokenswronglogic() 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);
        uint256 expected = 10e18;
        poolToken.approve(address(pool), 200e18);
        uint256 inputAmount = pool.getInputAmountBasedOnOutput(
            expected,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );
        pool.sellPoolTokens(expected);
        assert(inputAmount > expected);
        vm.stopPrank();
    }
```

Recommended Mitigation: Replace `TswapPool::swapExactOutput` function call in `TswapPool::sellPoolTokens` with `TswapPool::swapExactInput`. See below:

Code

```

/**
     * @notice wrapper function to facilitate users selling pool tokens
         in exchange of WETH
     * @param poolTokenAmount amount of pool tokens to sell
     * @return wethAmount amount of WETH received by caller
     */
    function sellPoolTokens(
        uint256 poolTokenAmount,
+       uint256 minOutputAmount,
    ) external returns (uint256 wethAmount) {
        return
-            swapExactOutput(
-               i_poolToken,
-               i_wethToken,
-               poolTokenAmount,
-               uint64(block.timestamp)
-            );

+            swapExactInput(
+       i_poolToken,
+         poolTokenAmount,
+         i_wethToken,
+         minOutputAmount,
+        uint64(block.timestamp)
+     )
    }
```

Low

[L-1] Event Details not emitted correctly which leads to potential migration issues and misleading information to users

Description: The event is initialised as:

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

and emitted as:

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

Impact: Potential Migration Issues and emitted event can mislead users on tokens they have added to the pool

Proof of Concept: See description

Recommended Mitigation: Refactor emitted event to:

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