TSwap Protocol Audit Report

Version 1.0

GushALKDev

January 3, 2026

TSwap Protocol Audit Report

GushALKDev

January 3, 2026

Prepared by: GushALKDev Lead Auditors: - GushALKDev

Table of Contents

Protocol Summary

The TSwap protocol 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). TSwap 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

GushALKDev 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.

Executive Summary

The TSwap protocol 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). TSwap 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

Issues found

Findings

High

[H-1] Incorrect fee calculation in `getInputAmountBasedOnOutput` causes protocol to take 90% fee instead of 0.3%

IMPACT: High **LIKELIHOOD:** High

Description: The `TSwapPool::getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user needs to input to receive a specific amount of output tokens. However, there is a typo in the fee calculation.

The function uses `10000` and `997` to calculate the fee, implying a 0.3% fee.

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

However, `997 / 10000` results in `0.0997`, which means the multiplier is `99.7%` ?? Actually, the math is: Target: `inputAmount * 997 / 1000 = outputAmount` (roughly) Current: `inputAmount = (outputAmount * 10000)/ 997`

If we look at `getOutputAmountBasedOnInput`: `inputAmount * 997 / 1000` Here `getInputAmountBasedOnOutput` should be the inverse. `outputAmount * 1000 / 997`

The current implementation uses `10000` in the numerator: `outputAmount * 10000 / 997`

This results in the user being required to send ~10x more tokens than they should. Calculated fee: `1 - (997/10000)= 1 - 0.0997 = 0.9003` => 90.03% fee.

Impact: Users are charged a massively high fee (90%+) for swaps where they specify the exact output amount. This effectively steals user funds or makes the protocol completely unusable for this transaction type.

Proof of Concept:

PoC

```
    function testFlawedSwapExactOutput() public {
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit(initialLiquidity, 0, initialLiquidity, uint64(
            block.timestamp));
        vm.stopPrank();

        // User wants to swap 1 WETH
        address attacker = makeAddr("attacker");
        vm.startPrank(attacker);
        poolToken.mint(attacker, 100e18);
        poolToken.approve(address(pool), 100e18);

        uint256 valueToBuy = 1e18;
        uint256 expectedCost = 1103362165907421361; // ~1.103 token ~=
            10% more than 1 (normal for these reserves)
        // However, the flawed math will ask for much more

        pool.swapExactOutput(poolToken, weth, valueToBuy, uint64(block.
            timestamp));

        uint256 actualCost = 100e18 - poolToken.balanceOf(attacker);
        // It should be around 1e18, but it is ~11e18
        console.log("Actual cost: ", actualCost);
        assert(actualCost > 10e18); // It charges ~10x more
        vm.stopPrank();
    }
```

Recommended Mitigation: Replace `10000` with `1000`.

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

[H-2] `TSwapPool::sellPoolTokens` calls `swapExactOutput` with incorrect parameters

IMPACT: High **LIKELIHOOD:** High

Description: The `sellPoolTokens` function allows users to sell their liquidity tokens (pool tokens) for WETH. It calls `swapExactOutput`:

```
    function sellPoolTokens(
        uint256 poolTokenAmount
    ) external returns (uint256 wethAmount) {
        return
```

```
            swapExactOutput(
                i_poolToken,
                i_wethToken,
                poolTokenAmount,
                uint64(block.timestamp)
            );
    }
```

The `swapExactOutput` function expects the third argument to be `outputAmount`. However, `poolTokenAmount` is passed, which is the *input* amount (the amount of pool tokens being sold). This suggests the function intends to swap an exact *input* of pool tokens, not get an exact *output* of WETH matching the pool token amount.

Additionally, `swapExactOutput` calculates `inputAmount` based on `outputAmount`. By passing `poolTokenAmount` as `outputAmount`, the function tries to calculate how many pool tokens are needed to get `poolTokenAmount` of WETH, which is not the intended behavior of "selling X pool tokens".

Impact: The function logic is fundamentally flawed. It will attempt to swap for a specific amount of WETH rather than selling a specific amount of pool tokens. If the price isn't 1:1, this will result in unexpected behavior or revert. Users cannot sell the exact amount of pool tokens they want.

Proof of Concept:

PoC

```
    function testSellPoolTokens() public {
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit(initialLiquidity, 0, initialLiquidity, uint64(
            block.timestamp));
        vm.stopPrank();

        address user = makeAddr("user");
        vm.startPrank(user);
        poolToken.mint(user, 10e18);
        poolToken.approve(address(pool), 10e18);

        // User wants to sell 10 pool tokens
        // Expected behavior: User calls sellPoolTokens(10e18) and pays
             10e18 pool tokens
        // Actual behavior: check logs
        pool.sellPoolTokens(10e18);

        // The user effectively requested to BUY 10 WETH output,
            identifying how many pool tokens input needed
```

```
20          // If the price is 1:1, it might work by coincidence, but if
                reserves skew, it breaks.
21          // Also, notice the function sellPoolTokens signature implies
                inputting a specific amount of pool tokens.
22          vm.stopPrank();
23      }
```

Recommended Mitigation: Use `swapExactInput` instead of `swapExactOutput` and pass `poolTokenAmount` as the `inputAmount`.

```
1       function sellPoolTokens(
2           uint256 poolTokenAmount
3       ) external returns (uint256 wethAmount) {
4   -       return swapExactOutput(i_poolToken, i_wethToken,
        poolTokenAmount, uint64(block.timestamp));
5   +       return swapExactInput(i_poolToken, poolTokenAmount, i_wethToken
        , minOutputAmount, uint64(block.timestamp));
6       }
```

Note: `sellPoolTokens` would need an additional `minOutputAmount` parameter to be safe.

[H-3] `TSwapPool::swapExactOutput` lacks slippage protection

IMPACT: High **LIKELIHOOD:** Medium

Description: The `swapExactOutput` function allows users to specify an exact amount of tokens they want to receive. However, it does not allow users to specify a maximum amount of input tokens they are willing to spend (`maxInputAmount`).

```
1       function swapExactOutput(
2           IERC20 inputToken,
3           IERC20 outputToken,
4           uint256 outputAmount,
5           uint64 deadline
6       )
```

Impact: If market conditions change (slippage) or if the pool state is manipulated before the transaction is executed, the user might end up modifying their input token balance significantly more than expected. They have no guarantee on the “price” they are paying for the output.

Proof of Concept:

PoC

```
1       function testSwapExactOutputNoSlippage() public {
2           uint256 initialLiquidity = 100e18;
3           vm.startPrank(liquidityProvider);
4           weth.approve(address(pool), initialLiquidity);
```

```
        poolToken.approve(address(pool), initialLiquidity);
        pool.deposit(initialLiquidity, 0, initialLiquidity, uint64(
            block.timestamp));
        vm.stopPrank();

        address user = makeAddr("user");
        vm.startPrank(user);
        poolToken.mint(user, 1000e18); // Rich user
        poolToken.approve(address(pool), 1000e18);

        // Normal swap expected cost
        uint256 outputToBuy = 1e18;

        // Imagine a front-runner comes in and changes the ratio
            drastically
        // (Just simulating a changed state here)
        vm.stopPrank();
        vm.startPrank(liquidityProvider);
        pool.withdraw(90e18, 90e18, 90e18, uint64(block.timestamp)); //
             Remove liquidity, price moves or depth drops
        vm.stopPrank();

        vm.startPrank(user);
        // User executes swap expecting previous price, but there is no
             maxInput param to revert
        pool.swapExactOutput(poolToken, weth, outputToBuy, uint64(block
            .timestamp));
        // Transaction succeeds but user paid way more than they might
            have authorized
        vm.stopPrank();
    }
```

Recommended Mitigation: Add a `maxInputAmount` parameter and require the calculated `inputAmount` to be less than or equal to `maxInputAmount`.

```
    function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxInputAmount,
        uint64 deadline
    )
    ...
+   if (inputAmount > maxInputAmount) {
+       revert TSwapPool__InputTooHigh(inputAmount, maxInputAmount);
+   }
```

[H-4] `TSwapPool::deposit` Missing Deadline Check

IMPACT: High **LIKELIHOOD:** Medium

Description: The `deposit` function accepts a `deadline` parameter but never uses it. The `revertIfDeadlinePassed` modifier is missing from the function signature.

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
```

Impact: Users who submit a deposit transaction with a deadline expect the transaction to fail if it stays pending for too long (e.g., to avoid depositing during unfavorable market/gas conditions). Without the check, the transaction could be executed at an arbitrary later time, potentially harming the user.

Proof of Concept:

PoC

```
    function testDepositDeadlinePassed() public {
        uint256 initialLiquidity = 100e18;
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), initialLiquidity);
        poolToken.approve(address(pool), initialLiquidity);

        // Deadline is in the past
        uint64 pastDeadline = uint64(block.timestamp - 1);

        // Should revert, but it succeeds
        pool.deposit(initialLiquidity, 0, initialLiquidity,
            pastDeadline);

        vm.stopPrank();
        assertEq(pool.totalSupply(), initialLiquidity);
    }
```

Recommended Mitigation: Add the `revertIfDeadlinePassed` modifier.

```
    function deposit(
        ...
        uint64 deadline
    )
        external
+       revertIfDeadlinePassed(deadline)
```

```
7           revertIfZero(wethToDeposit)
8           returns (uint256 liquidityTokensToMint)
```

Medium

[M-1] Fee-on-transfer logic breaks protocol invariant (if implemented)

IMPACT: High **LIKELIHOOD:** Low

Description: In `TSwapPool::_swap`, there is some logic regarding an extra token incentive:

```
1 -         if (swap_count >= SWAP_COUNT_MAX) {
2 -             swap_count = 0;
3 -             outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
4 -         }
```

Sending tokens out of the contract without a corresponding swap input disrupts the `x * y = k` invariant. The contract's balance of `outputToken` decreases without `inputToken` increasing to compensate.

Impact: If this logic were active, the constant product invariant would be broken, leading to incorrect pricing for subsequent swaps and potential depletion of the pool over time.

Recommended Mitigation: Do not implement fee-on-transfer mechanics that simply remove tokens from the pool's reserves involved in the pricing curve. If incentives are needed, they should be funded separately or accounted for mathematically.

Low

[L-1] `LiquidityAdded` event parameters are mismatched

IMPACT: Low **LIKELIHOOD:** High

Description: In `TSwapPool::_addLiquidityMintAndTransfer`, the `LiquidityAdded` event is emitted with the parameters in the wrong order.

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

The event definition is:

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

The call swaps `wethDeposited` and `poolTokensDeposited`.

Impact: Off-chain indexers and UIs will display incorrect deposit amounts for WETH and PoolTokens, confusing users.

Recommended Mitigation: Swap the arguments in the emit statement.

[L-2] `TSwapPool::swapExactInput` returns unused value

IMPACT: Low **LIKELIHOOD:** High

Description: The `swapExactInput` function returns `uint256 output`, but the return value is not well-documented and the function ends with `_swap` which doesn't return the value explicitly in a `return` statement (though solidity handles named returns).

```
1     function swapExactInput(
2         IERC20 inputToken,
3         uint256 inputAmount,
4         IERC20 outputToken,
5         uint256 minOutputAmount,
6         uint64 deadline
7     )
8         public
9         revertIfZero(inputAmount)
10        revertIfDeadlinePassed(deadline)
11        returns (uint256 output)
12    {
13        uint256 inputReserves = inputToken.balanceOf(address(this));
14        uint256 outputReserves = outputToken.balanceOf(address(this));
15
16        uint256 outputAmount = getOutputAmountBasedOnInput(
17            inputAmount,
18            inputReserves,
19            outputReserves
20        );
21
22        if (outputAmount < minOutputAmount) {
23            revert TSwapPool__OutputTooLow(outputAmount,
                  minOutputAmount);
24        }
25
26        _swap(inputToken, inputAmount, outputToken, outputAmount);
27    }
```

Impact: Confusing API for integrators.

Recommended Mitigation: Ensure the return value is explicit and documented.

[L-3] `PoolFactory::createPool` uses wrong token symbol

IMPACT: Low **LIKELIHOOD:** High

Description: When creating a new pool, the `liquidityTokenSymbol` is created using the name of the token instead of the symbol.

```
1 string memory liquidityTokenSymbol = string.concat("ts", IERC20(
      tokenAddress).name());
```

Impact: The symbol of the LP token will be incorrect (e.g., “tsWrapped Ether” instead of “tsWETH”), which is confusing.

Recommended Mitigation: Use `.symbol()` instead of `.name()`.

Informational

[I-1] Missing `zero address` checks

Description: Several functions and constructors lack checks for `address(0)`.

- `PoolFactory::constructor: wethToken`

```
1     constructor(address wethToken) {
2         i_wethToken = wethToken;
3     }
```

- `PoolFactory::createPool: tokenAddress`

```
1     function createPool(address tokenAddress) external returns (address
          ) {
2         if (s_pools[tokenAddress] != address(0)) {
3             revert PoolFactory__PoolAlreadyExists(tokenAddress);
4         }
5         // ...
```

- `TSwapPool::constructor: wethToken, poolToken`

```
1     constructor(
2         address poolToken,
3         address wethToken,
4         string memory liquidityTokenName,
5         string memory liquidityTokenSymbol
6     ) ERC20(liquidityTokenName, liquidityTokenSymbol) {
7         i_wethToken = IERC20(wethToken);
8         i_poolToken = IERC20(poolToken);
```

```
9       }
```

Recommended Mitigation: Add `require(address != address(0))` checks.

[I-2] Missing `revertIfZero` checks

Description: Several functions accept amount parameters that should be checked against 0 to avoid wasted gas or logic errors.

- `TSwapPool::deposit`: `maximumPoolTokensToDeposit`

```
1       function deposit(
2           uint256 wethToDeposit,
3           uint256 minimumLiquidityTokensToMint,
4           uint256 maximumPoolTokensToDeposit,
5           uint64 deadline
6       )
7           external
8           revertIfZero(wethToDeposit)
9           returns (uint256 liquidityTokensToMint)
10      {
11          // ...
```

- `TSwapPool::swapExactInput`: `minOutputAmount`

```
1       function swapExactInput(
2           IERC20 inputToken,
3           uint256 inputAmount,
4           IERC20 outputToken,
5           uint256 minOutputAmount,
6           uint64 deadline
7       )
8           public
9           revertIfZero(inputAmount)
10          revertIfDeadlinePassed(deadline)
11          returns (uint256 output)
12      {
13          // ...
```

Recommended Mitigation: Add `revertIfZero` modifier or require checks.

[I-3] Missing `indexed` event fields

Description: Events `PoolCreated`, `LiquidityAdded`, `LiquidityRemoved`, and `Swap` do not use `indexed` keywords on addresses (except `LiquidityAdded` which has one). `Swap` is missing indexed keys for `tokenIn` and `tokenOut`, making it hard to filter swaps by token.

Found in src/TSwapPool.sol:

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

    event LiquidityRemoved(
        address indexed liquidityProvider,
        uint256 wethWithdrawn,
        uint256 poolTokensWithdrawn
    );

    event Swap(
        address indexed swapper,
        IERC20 tokenIn,
        uint256 amountTokenIn,
        IERC20 tokenOut,
        uint256 amountTokenOut
    );
```

Found in src/PoolFactory.sol:

```
    event PoolCreated(address tokenAddress, address poolAddress);
```

Recommended Mitigation: Add `indexed` to address parameters in events.

[I-4] Magic Numbers

Description: The codebase uses literal numbers like `1000`, `997`, `10000`, `1e18`.

Found in:

In src/TSwapPool.sol:

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

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

```
        return
            getOutputAmountBasedOnInput(
                1e18,
                i_wethToken.balanceOf(address(this)),
```

```
                i_poolToken.balanceOf(address(this))
            );
```

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

Recommended Mitigation: Define these as `constant` variables (e.g., `FEE_MULTIPLIER = 997`).

[I-5] PUSH0 Opscode capability

Description: The contract uses `pragma solidity 0.8.20`, which may use the `PUSH0` opcode. This opcode is not supported on all EVM chains (e.g., possibly L2s or older mainnet forks depending on timing).

Found in `src/TSwapPool.sol` and `src/PoolFactory.sol`:

```
pragma solidity 0.8.20;
```

Recommended Mitigation: Verify target chain compatibility or use 0.8.19.

[I-6] Unused Custom Errors

Description: `PoolFactory__PoolDoesNotExist` is defined but never used.

Found in `src/PoolFactory.sol`:

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

Recommended Mitigation: Remove unused errors.

[I-7] `PoolFactory`: Liquidity token name missing zero length check

Description: The `PoolFactory::createPool` function creates a new liquidity token with a name derived from the token's name. However, there is no check to ensure that the liquidity token name is not empty or zero length.

Found in `src/PoolFactory.sol`:

```
string memory liquidityTokenName = string.concat("T-Swap ", IERC20(
    tokenAddress).name());
```

Recommended Mitigation: Add a check to ensure `liquidityTokenName` is not empty.

[I-8] `TSwapPool::deposit`: Error emits constant value

Description: The error `TSwapPool__WethDepositAmountTooLow` emits `MINIMUM_WETH_LIQUIDITY`, which is a constant. Emitting constants in errors is unnecessary and increases gas costs.

Found in `src/TSwapPool.sol`:

```
revert TSwapPool__WethDepositAmountTooLow(
    MINIMUM_WETH_LIQUIDITY,
    wethToDeposit
);
```

Recommended Mitigation: Remove the constant from the error definition and emission.

[I-9] `TSwapPool::deposit`: Unused local variable `poolTokenReserves`

Description: The `poolTokenReserves` variable is defined but never used in the `deposit` function.

Found in `src/TSwapPool.sol`:

```
uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

Recommended Mitigation: Remove the unused variable.

[I-10] `TSwapPool::swapExactInput`: Function should be `external`

Description: The `swapExactInput` function is marked as **`public`** but is not called internally. It should be marked as `external` to save gas.

Found in `src/TSwapPool.sol`:

```
function swapExactInput(
    ...
)
    public
```

Recommended Mitigation: Change visibility to `external`.

[I-11] `TSwapPool::swapExactInput`: Missing NatSpec

Description: The `swapExactInput` function lacks NatSpec documentation, making it difficult for developers and auditors to understand its purpose and parameters.

Found in `src/TSwapPool.sol`:

```
1 function swapExactInput(
```

Recommended Mitigation: Add complete NatSpec documentation.

[I-12] `TSwapPool::swapExactOutput`: Missing `deadline` in NatSpec

Description: The NatSpec for `swapExactOutput` is missing the `@param deadline` documentation.

Found in `src/TSwapPool.sol`:

```
1 function swapExactOutput(
```

Recommended Mitigation: Add the missing parameter documentation.

[I-13] `TSwapPool::swapExactOutput`: Missing `maxInputAmount` indication

Description: The function does not clearly indicate or enforce a maximum input amount, which is crucial for slippage protection (related to H-3).

Recommended Mitigation: Implement `maxInputAmount` parameter (as recommended in H-3).

[I-14] `TSwapPool::_swap`: Invariant check missing

Description: The `_swap` function does not verify that the constant product invariant (`x * y = k`) holds after the swap.

Recommended Mitigation: Consider adding an invariant check in debug/testing mode or ensure math guarantees it.