TSwap Audit Report

Version 1.0

https://github.com/Heavens01

July 25, 2025

TSwap Audit Report

Heavens01

July 25, 2025

Prepared by: Heavens01 Lead Security Researcher(s): - Heavens01

Table of Contents

Protocol Summary

TSwap

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

TSwap Pools

The protocol starts as simply a `PoolFactory` contract. This contract is used to create new "pools" of tokens. It helps make sure every pool token uses the correct logic. But all the magic is in each `TSwapPool` contract.

You can think of each `TSwapPool` contract as it's own exchange between exactly 2 assets. Any ERC20 and the WETH token. These pools allow users to permissionlessly swap between an ERC20 that has a pool and WETH. Once enough pools are created, users can easily "hop" between supported ERC20s.

For example: 1. User A has 10 USDC 2. They want to use it to buy DAI 3. They `swap` their 10 USDC -> WETH in the USDC/WETH pool 4. Then they `swap` their WETH -> DAI in the DAI/WETH pool

Every pool is a pair of `TOKEN X` & `WETH`.

There are 2 functions users can call to swap tokens in the pool. - `swapExactInput` - `swapExactOutput`

We will talk about what those do in a little.

Liquidity Providers

In order for the system to work, users have to provide liquidity, aka, "add tokens into the pool".

Why would I want to add tokens to the pool?

The TSwap protocol accrues fees from users who make swaps. Every swap has a `0.3` fee, represented in `getInputAmountBasedOnOutput` and `getOutputAmountBasedOnInput`. Each applies a `997` out of `1000` multiplier. That fee stays in the protocol.

When you deposit tokens into the protocol, you are rewarded with an LP token. You'll notice `TSwapPool` inherits the `ERC20` contract. This is because the `TSwapPool` gives out an ERC20 when Liquidity Providers (LP)s deposit tokens. This represents their share of the pool, how much they put in. When users swap funds, 0.03% of the swap stays in the pool, netting LPs a small profit.

LP Example

1. LP A adds 1,000 WETH & 1,000 USDC to the USDC/WETH pool
    1. They gain 1,000 LP tokens
2. LP B adds 500 WETH & 500 USDC to the USDC/WETH pool
    1. They gain 500 LP tokens
3. There are now 1,500 WETH & 1,500 USDC in the pool
4. User A swaps 100 USDC -> 100 WETH.
    1. The pool takes 0.3%, aka 0.3 USDC.
    2. The pool balance is now 1,400.3 WETH & 1,600 USDC
    3. aka: They send the pool 100 USDC, and the pool sends them 99.7 WETH

Note, in practice, the pool would have slightly different values than 1,400.3 WETH & 1,600 USDC due to the math below.

Core Invariant

Our system works because the ratio of Token A & WETH will always stay the same. Well, for the most part. Since we add fees, our invariant technially increases.

- `x * y = k`
- x = Token Balance X
- y = Token Balance Y
- k = The constant ratio between X & Y

```
y = Token Balance Y
x = Token Balance X
x * y = k
x * y = (x + Cx) * (y - Cy)
Cx = Change of token balance X
Cy = Change of token balance Y
$\beta$ = (Cy / y)
$\alpha$ = (Cx / x)

```

```
Final invariant equation without fees:
Cx = ($\beta$/(1-$\beta$)) * x
Cy = ($\alpha$/(1+$\alpha$)) * y

Invariant with fees:
$\rho$ = fee (between 0 & 1, aka a percentage)
$\gamma$ = (1 - $\rho$)
Cx = ($\beta$/(1-$\beta$)) * (1/$\gamma$) * x
Cy = ($\alpha$$\gamma$/(1+$\alpha$$\gamma$)) * y
```

Our protocol should always follow this invariant in order to keep swapping correctly!

Make a swap

After a pool has liquidity, there are 2 functions users can call to swap tokens in the pool. - `swapExactInput` - `swapExactOutput`

A user can either choose exactly how much to input (ie: I want to use 10 USDC to get however much WETH the market says it is), or they can choose exactly how much they want to get out (ie: I want to get 10 WETH from however much USDC the market says it is.

This codebase is based loosely on Uniswap v1

Disclaimer

The HEAVENS01 team 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
	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
```

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

*We spent 1 Week (7 Days) with one auditor using solidity metrics tool, aderyn, slither, stateful fuzzing, stateless fuzzing and manual reading of codebase.

Issues found

Severity	Number of issues found
High	7
Medium	1
Low	3
Info/Gas	Not Included

Severity	Number of issues found
Total	13

Findings

High

[H-1] No Access Control on Pool Creation and First Deposit (Unrestricted Access + Permanent Price Manipulation)

Description:

The `PoolFactory::createPool` function allows anyone to create a pool for any poolToken (ERC20 Tokens), and the `TSwapPool::deposit` function permits the first depositor to set the initial WETH-to-pool token ratio without access control. Additionally, once a pool is created for a poolToken, no new pool can be created for the same token, locking in the initial ratio set by a potentially malicious user.

Impact:

- Malicious users can create pools and set unfavorable WETH-to-pool token ratios, manipulating prices to their advantage.
- Future liquidity providers are forced to follow the initial ratio, potentially leading to financial losses or reduced pool attractiveness.
- Since createPool prevents duplicate pools for the same poolToken, a maliciously set ratio cannot be corrected by creating a new pool, critically damaging the protocol's usability and trust.
- Uncontrolled pool creation could enable spam or malicious pools, undermining protocol integrity.

Proof of Concept:

In `PoolFactory.sol`, `createPool` lacks access control and prevents duplicate pools:

```
    function createPool(address tokenAddress) external returns (address
        ) {
        if (s_pools[tokenAddress] != address(0)) {
            revert PoolFactory__PoolAlreadyExists(tokenAddress);
        }
        // ... pool creation logic
    }
```

In `TSwapPool.sol::deposit`, the first deposit sets the ratio without restrictions:

```
function deposit(uint256 wethToDeposit, uint256
    minimumLiquidityTokensToMint, uint256 maximumPoolTokensToDeposit,
    uint64 deadline) external revertIfZero(wethToDeposit) returns (
    uint256 liquidityTokensToMint) {
    // ...Checks
    if (totalLiquidityTokenSupply() > 0) {
        // ... ratio-based deposit logic
    } else {
        _addLiquidityMintAndTransfer(wethToDeposit,
            maximumPoolTokensToDeposit, wethToDeposit);
        liquidityTokensToMint = wethToDeposit;
    }
}
```

The first depositor’s `wethToDeposit` and `maximumPoolTokensToDeposit` define the pool’s price ratio, which cannot be corrected due to the `PoolAlreadyExists` check.

Appending the following test in `PoolFactoryTest.t.sol` demonstrates the issue:

POC Test

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

function testUnrestrictedFirstDepositSetsPermanentPrice() public {
    address maliciousUser = makeAddr("maliciousUser");
    vm.startPrank(maliciousUser);
    address poolAddress = factory.createPool(address(tokenA));
    TSwapPool pool = TSwapPool(poolAddress);
    tokenA.mint(maliciousUser, 1000e18);
    mockWeth.mint(maliciousUser, 1e18);
    tokenA.approve(poolAddress, 1000e18);
    mockWeth.approve(poolAddress, 1e18);
    pool.deposit(1e18, 1e18, 1000e18, uint64(block.timestamp));
    // Malicious user sets 1 WETH = 1000 tokenA ratio
    assert(pool.getPriceOfOneWethInPoolTokens() <= 1000e18); // (<=)
        symbol due to fees
    vm.stopPrank();
    // Attempt to create new pool for tokenA fails
    vm.expectRevert(abi.encodeWithSelector(PoolFactory.
        PoolFactory__PoolAlreadyExists.selector, address(tokenA)));
    factory.createPool(address(tokenA));
}
```

Recommended Mitigation:

Restrict `PoolFacory::createPool` pool creation and first deposit made in pool `TSwapPool::deposit` to authorized addresses using Governance: * Implement governance to the pool using this guide by openzeppelin: (https://docs.openzeppelin.com/contracts/5.x/api/governance)

[H-2] Unrestricted Pool Creation for WETH and Liquidity Tokens (Improper Input Validation + Protocol Misuse, WETH Recycling and Repeated Exploitation)

Description:

The `PoolFactory::createPool` function allows pools to be created for the WETH token or liquidity pool (LP) tokens from other pools. This is unintended, as WETH is meant to be a paired token, and LP tokens represent pool shares, not independent assets. A user can exploit this by creating a pool with an LP token as the poolToken, pairing it with WETH, swapping gained LP tokens for WETH, and reusing the WETH to deposit into the original or other pools, reducing the cost of deposits and misrepresenting deposit sizes.

Impact:

- Creating a pool with WETH as the `poolToken` creates redundant WETH-WETH pairs, causing confusion.
- Creating a pool with an LP token allows users to:
  - Swap new LP tokens for WETH in the nested pool, then redeposit the WETH into the original or another pool, requiring only the `poolToken` from their funds, lowering the deposit cost.
  - Repeat this process, accumulating LP tokens at a reduced cost, which misrepresents their actual deposit size and undermines the protocol's design.
- Such pools enable manipulative strategies, reduce protocol trust, and disrupt the economic model by allowing users to gain disproportionate control over liquidity.

Proof of Concept:

In `PoolFactory.sol`, `createPool` does not restrict the `tokenAddress`:

```
function createPool(address tokenAddress) external returns (address) {
    // No checks for tokenAddress value
    if (s_pools[tokenAddress] != address(0)) {
        revert PoolFactory__PoolAlreadyExists(tokenAddress);
    }
    // ... pool creation logic
}
```

This allows `tokenAddress` to Court of Appeal: be WETH or an LP token (i.e., a `TSwapPool` address). A user can create a pool with an LP token, deposit to gain new LP tokens, swap for WETH, and redeposit WETH, reducing their cost.

Appending the following test in `PoolFactoryTest.t.sol` demonstrates the issue:

POC Test:

```
function testLpTokenPoolExploitation() public {
    // Create initial pool for tokenA
    address tokenAPool = factory.createPool(address(tokenA));
    TSwapPool poolA = TSwapPool(tokenAPool);
    address liquidityProvider = makeAddr("liquidityProvider");
    vm.startPrank(liquidityProvider);
    tokenA.mint(liquidityProvider, 100e18);
    mockWeth.mint(liquidityProvider, 100e18);
    tokenA.approve(tokenAPool, 100e18);
    mockWeth.approve(tokenAPool, 100e18);
    poolA.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));

    // Create pool with tokenA's LP token as poolToken
    address lpTokenPool = factory.createPool(address(poolA));
    TSwapPool lpPool = TSwapPool(lpTokenPool);
    poolA.approve(lpTokenPool, 50e18);
    mockWeth.mint(liquidityProvider, 50e18);
    mockWeth.approve(lpTokenPool, 50e18);
    lpPool.deposit(50e18, 50e18, 50e18, uint64(block.timestamp));

    // Swap LP tokens for WETH in lpTokenPool
    poolA.approve(lpTokenPool, 25e18);
    //Note: 15e18 WETH minimum expected due to wrong implementation of fees
    lpPool.swapExactInput(poolA, 25e18, mockWeth, 15e18, uint64(block.timestamp));

    // Redeposit gained WETH into tokenA pool
    tokenA.mint(liquidityProvider, 20e18);
    tokenA.approve(tokenAPool, 20e18);
    mockWeth.approve(tokenAPool, 15e18);
    poolA.deposit(15e18, 15e18, 15e18, uint64(block.timestamp));
    // User gained additional LP tokens with reduced WETH cost
    assert(poolA.balanceOf(liquidityProvider) > 15e18);
    vm.stopPrank();
}
```

Recommended Mitigation:

Add validation in `PoolFactory::createPool` to prevent pools for WETH or LP tokens:

```
contract PoolFactory {
+   error PoolFactory__InvalidTokenAddress();

    function createPool(address tokenAddress) external returns (address
        ) {
+       if (tokenAddress == address(0) || tokenAddress == i_wethToken
    || TSwapPool(tokenAddress).totalSupply.selector != bytes4(0)) {
+           revert PoolFactory__InvalidTokenAddress();
+       }
        if (s_pools[tokenAddress] != address(0)) {
            revert PoolFactory__PoolAlreadyExists(tokenAddress);
        }
        // ... pool creation logic
    }
}
```

[H-3] Protocol Invariant Broken by Extra WETH Transfer After Tenth Swap (Tenth Swap Free Transfer + Loss of Liquidity Unaccounted For)

Description:

The `TSwapPool::_swap` function breaks the constant product invariant (`x * y = k`) by transferring an additional 1 WETH (1e18) to msg.sender after every tenth swap, without adjusting the pool's reserves or accounting for this transfer in the swap calculations. This disrupts the pool's economic balance and violates the invariant that the product of WETH and pool token reserves remains constant.

Impact:

The extra 1 WETH transfer after the tenth swap reduces the pool's WETH reserves without a corresponding increase in pool tokens, breaking the constant product invariant. This allows users to extract value from the pool without contributing equivalent assets, potentially draining liquidity, destabilizing the pool's pricing, and reducing returns for liquidity providers, undermining the protocol's integrity and fairness.

Proof of Concept:

In `TSwapPool.sol`, the `_swap` function includes a swap count and transfer:

```
    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);
    }
```

Appending this test in `TSwapPool.t.sol` demonstrates the issue:

```
function testSwapBreaksInvariantAfterTenthSwap() 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);
    poolToken.mint(user, 10e18);
    poolToken.approve(address(pool), 10e18);
    uint256 initialK = weth.balanceOf(address(pool)) * poolToken.balanceOf(address(pool));

    // Perform 10 swaps to trigger extra WETH transfer
    for (uint256 i = 0; i < 10; i++) {
        pool.swapExactInput(poolToken, 1e18, weth, 0, uint64(block.timestamp));
    }

    // Check invariant after 10th swap
    uint256 finalK = weth.balanceOf(address(pool)) * poolToken.balanceOf(address(pool));
    assert(finalK < initialK); // Invariant broken due to extra 1 WETH transfer
    vm.stopPrank();
}
```

Recommended Mitigation:

Remove the extra WETH transfer to maintain the constant product invariant:

```
contract TSwapPool is ERC20 {
    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);
    }
}
```

[H-4] Deadline Parameter Not Enforced in `TSwapPool::deposit` (Improper Transaction Validation + Potential Stale Execution)

Description:

The `TSwapPool::deposit` function accepts a `deadline` parameter but does not enforce it with the `revertIfDeadlinePassed` modifier. This allows transactions queued in the mempool to execute even if the deadline has passed, potentially leading to unintended executions under changed market conditions.

Impact:

- Users may have deposits executed at unfavorable prices if mined after the deadline, risking financial losses.
- Lack of deadline enforcement undermines user control over transaction timing, reducing protocol trust.
- Stale transactions could disrupt pool ratios or liquidity expectations.

Proof of Concept:

In `TSwapPool.sol`, the `deposit` function lacks the `revertIfDeadlinePassed` modifier:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    uint64 deadline
) external revertIfZero(wethToDeposit) /*No revertIfDeadlineHasPassed*/
    returns (uint256 liquidityTokensToMint) {
    // ... deposit logic
}
```

No check reverts if `block.timestamp > deadline`, unlike functions like `withdraw` or `swapExactInput`.

Appending this test in `TSwapPool.t.sol` demonstrates the issue:

Proof Of Code:

```
function testDepositIgnoresDeadline() public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    // Set deadline to a past timestamp
    uint64 pastDeadline = uint64(block.timestamp);
    // Move time forward to simulate a past deadline
    vm.warp(pastDeadline + 100);
    vm.roll(block.number + 1);
    // Deposit should revert but does not
    pool.deposit(100e18, 100e18, 100e18, pastDeadline);
    assertEq(pool.balanceOf(liquidityProvider), 100e18);
    vm.stopPrank();
}
```

Recommended Mitigation:

Add the `revertIfDeadlinePassed` modifier to the `deposit` function:

```
contract TSwapPool is ERC20 {
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
-    ) external revertIfZero(wethToDeposit) returns (uint256
    liquidityTokensToMint) {
```

```
+    ) external revertIfZero(wethToDeposit) revertIfDeadlinePassed(
    deadline) returns (uint256 liquidityTokensToMint) {
         // ... deposit logic
     }
 }
```

[H-5] Incorrect Fee Calculation in `TSwapPool::getOutputAmountBasedOnInput` (Overcharging Fees + Reduced User Output)

Description:

The `TSwapPool::getOutputAmountBasedOnInput` function incorrectly calculates the fee for swaps, charging a higher fee than the intended 0.3% (3/1000). The test reveals that for an input of 10e18 WETH, the output is 18.132e18 pool tokens instead of the expected 19.94e18 pool tokens (20e18 minus a 0.3% fee of 0.06e18), indicating an overcharge in fees.

Impact:

The incorrect fee calculation in `getOutputAmountBasedOnInput` results in users receiving significantly fewer output tokens than expected, reducing their returns and making swaps less attractive and overly expensive. This overcharging undermines the protocol's fairness, potentially driving users away and affecting trust and adoption, as they are unfairly penalized with higher-than-advertised fees during swaps.

Proof of Concept:

Appending this test in `TSwapPool.t.sol` demonstrates the issue:

```
function testGetOutputAmountBasedOnInputFee() public {
    // Swapping 10e18 WETH for pool tokens with a fee
    // Calculate output for 10e18 WETH input
    uint256 inputAmount = 10e18; // Output amount without fee should be
         20e18 pool tokens
    uint256 inputReserves = 100e18; // WETH
    uint256 outputReserves = 200e18; // Pool tokens (i.e 1 weth = 2
        pool tokens)
    uint256 expectedfeeinWETH = 3e16; // 0.3% of 10e18 of WETH
    uint256 expectedFeeInPoolTokens = 6e16; // 0.3% of Pool Tokens (as
        1 weth = 2 pool tokens)
    uint256 outputAmountWithoutFee = 20e18; // inputAmount * 2 (1 WETH
        = 2 Pool Tokens)

    uint256 outputAmount = pool.getOutputAmountBasedOnInput(inputAmount
        , inputReserves, outputReserves);
    console.log("Output Amount (Received Pool Token): ", outputAmount);
    // Assert the expected fee is what was deducted in pool tokens
    assertEq(outputAmount, outputAmountWithoutFee -
        expectedFeeInPoolTokens);
    vm.stopPrank();
}
```

Test output shows: * Input: 10e18 WETH * Expected output: 19.94e18 pool tokens (20e18 - 0.06e18 fee) * Actual output: 18.132e18 pool tokens * The test fails because the function deducts a higher fee (approximately 1.868e18 pool tokens) instead of the intended 0.06e18, overcharging users.

Recommended Mitigation:

Correct the fee calculation in `getOutputAmountBasedOnInput` to apply a 0.3% fee accurately:

```diff
contract TSwapPool is ERC20 {
    function getOutputAmountBasedOnInput(
        uint256 inputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(inputAmount)
        revertIfZero(outputReserves)
        returns (uint256 outputAmount)
    {
        uint256 inputAmountMinusFee = inputAmount * 997;
        uint256 numerator = inputAmountMinusFee * outputReserves;
-       uint256 denominator = (inputReserves * 1000) +
    inputAmountMinusFee;
+       uint256 denominator = (inputReserves * 1000);
        return numerator / denominator;
    }
}
```

[H-6] Incorrect Fee Constants and Implementation in `TSwapPool::getInputAmountBasedOnOutput` (Overcharging Fees + Incorrect Input Calculation)

Description:

The `TSwapPool::getInputAmountBasedOnOutput` function uses incorrect fee constants (10000 in the numerator, 997 in the denominator) instead of 1000 and 997 to align with `getOutputAmountBasedOnInput` for a 0.3% fee. Additionally, the formula is incorrectly structured, leading to an overcharge. The test shows that for a 20e18 pool token output, the function requires 111.445e18 WETH instead of the expected 10.03e18 WETH, indicating an effective fee of ~11.144% instead of 0.3%.

Impact:

The incorrect fee constants and flawed formula in `getInputAmountBasedOnOutput` cause users to provide significantly more input tokens than expected, reducing swap profitability and undermining the protocol's fairness. This overcharging, far exceeding the advertised 0.3% fee, can deter users and erode trust, while the incorrect input calculation disrupts the pool's constant product invariant, potentially affecting economic balance.

Proof of Concept:

Appending this test in `TSwapPool.t.sol` demonstrates the issue:

```
function testGetInputAmountBasedOnOutputFee() public {
    // Requesting 20e18 pool tokens for WETH input
    uint256 outputAmount = 20e18; // Desired pool token output
    uint256 inputReserves = 100e18; // WETH
    uint256 outputReserves = 200e18; // Pool tokens (1 WETH = 2 pool
       tokens)
    uint256 expectedInputWithoutFee = 10e18; // 20e18 / 2
    uint256 expectedFeeInWETH = 3e16; // 0.3% of 10e18 WETH
    uint256 expectedInputWithFee = 10e18 + 3e16; // 10.03e18 WETH

    uint256 inputAmount = pool.getInputAmountBasedOnOutput(outputAmount
       , inputReserves, outputReserves);
    console.log("Input Amount (Required WETH): ", inputAmount);
    // Assert the expected input includes 0.3% fee
    assertEq(inputAmount, expectedInputWithFee);
}
```

Test output shows: * Output: 20e18 pool tokens * Expected input: 10.03e18 WETH (10e18 + 0.3% fee of 0.03e18) * Actual input: 111.445e18 WETH * The test fails because the function overcharges, requiring ~11.144% more WETH than intended.

Recommended Mitigation:

Correct the fee constants and formula in `getInputAmountBasedOnOutput` to apply a 0.3% fee and maintain the constant product invariant:

```
contract TSwapPool is ERC20 {
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-       return ((inputReserves * outputAmount) * 10000) / ((
    outputReserves - outputAmount) * 997);
+       return ((inputReserves * outputAmount) * 1000) / ((
    outputReserves) * 997);
    }
}
```

[H-7] Issues in TSwapPool::sellPoolTokens Function (Missing Slippage Protection, Incorrect Function Call, Unsafe Deadline + Potential Financial Loss For Protocol/User)

Description:

The `TSwapPool::sellPoolTokens` function has multiple issues: it lacks a `minWethAmount` parameter for slippage protection, incorrectly calls `swapExactOutput` instead of `swapExactInput` despite intending to swap a fixed input of pool tokens. These flaws make the function unsafe and unreliable for users.

Impact:

The absence of a `minWethAmount` parameter exposes users to unbounded slippage, potentially receiving far less WETH than expected in volatile or low-liquidity pools. The incorrect use of `swapExactOutput` (instead of `swapExactInput`) misaligns with the function's intent to swap a fixed `poolTokenAmount`, leading to incorrect swap logic and unpredictable outcomes.

Proof of Concept:

In `TSwapPool.sol`, the `sellPoolTokens` function is flawed:

```
function sellPoolTokens(uint256 poolTokenAmount) external returns (
    uint256 wethAmount) {
    return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount,
        uint64(block.timestamp));
}
```

- No `minWethAmount` parameter allows excessive slippage.
- `swapExactOutput` is used, treating `poolTokenAmount` as the output, which contradicts the intent to input a fixed `poolTokenAmount`. Appending this test in `TSwapPool.t.sol` demonstrates the issues:
  Proof Of Code:

```
function testSellPoolTokensTreatsPoolTokenAsOutput() public {
    // Setup pool with realistic liquidity
    vm.startPrank(liquidityProvider);
    weth.mint(liquidityProvider, 100e18);
    poolToken.mint(liquidityProvider, 100e18);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    address newUser = makeAddr("newUser");
    vm.startPrank(newUser);
    poolToken.mint(newUser, 120e18);
    poolToken.approve(address(pool), 120e18);

    // Test: sellPoolTokens treats poolTokenAmount as WETH output, not
        poolToken input
    uint256 initialWethBal = weth.balanceOf(newUser);
    uint256 intitialPoolTokenBal = poolToken.balanceOf(newUser);

    uint256 poolTokenAmount = 10e18; // User intends to input 10e18
        pool tokens
    uint256 inputTokensRequired = pool.sellPoolTokens(poolTokenAmount);

    uint256 finalWethBal = weth.balanceOf(newUser);
    uint256 finalPoolTokenBal = poolToken.balanceOf(newUser);

    // Weth balance is now 10e18, as poolTokenAmount is treated as WETH
         output
    assertEq(initialWethBal, 0);
    assertEq(finalWethBal, poolTokenAmount);
    // Pool token sold is now higher than intended due to the
        swapExactOutput arrangement and fee
    assertLt(finalPoolTokenBal, intitialPoolTokenBal - poolTokenAmount)
        ;
    assertEq(finalPoolTokenBal, intitialPoolTokenBal -
        inputTokensRequired);
    vm.stopPrank();
}
```

Recommended Mitigation:

Add a `minWethAmount` parameter, use `swapExactInput`:

```
contract TSwapPool is ERC20 {
+   error TSwapPool__OutputTooLow(uint256 actual, uint256 min);

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

MEDIUM

[M-1] Missing maxInputAmount Parameter in `TSwapPool::swapExactOutput` (Lack of Slippage Protection + Potential Financial Loss)

Description:

The `TSwapPool::swapExactOutput` function lacks a `maxInputAmount` parameter to limit the input tokens a user is willing to spend. Without this, users have no control over slippage, and the function may consume an unexpectedly high amount of input tokens to achieve the desired output, leading to potential financial losses.

Impact:

The absence of a `maxInputAmount` parameter in `TSwapPool::swapExactOutput` exposes users to unbounded slippage, especially in low-liquidity or volatile pools, where the required input tokens could far exceed expectations. This can drain user balances or make swaps uneconomical, eroding trust in the protocol and discouraging its use by users and integrators who rely on predictable swap costs.

Proof of Concept:

In `TSwapPool.sol`, `swapExactOutput` does not include or check a `maxInputAmount` parameter:

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    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);
    _swap(inputToken, inputAmount, outputToken, outputAmount);
}
```

Appending this test in `TSwapPool.t.sol` demonstrates the issue:

Proof Of Code:

```
function testSwapExactOutputLacksSlippageProtection() public {
    // Setup pool with low liquidity
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 10e18); // Low WETH liquidity
    poolToken.approve(address(pool), 100e18);
    pool.deposit(10e18, 10e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    address slippageVictim = makeAddr("slippageVictim");
    vm.startPrank(slippageVictim);
    poolToken.mint(slippageVictim, 10_000e18);
    poolToken.approve(address(pool), 10_000e18);
    // User expects to spend ~11 pool tokens for 9 WETH, but input
        could be higher
    uint256 inputAmount = pool.swapExactOutput(poolToken, weth, 9e18,
        uint64(block.timestamp));
    // Without maxInputAmount, inputAmount could be unexpectedly large
        (e.g., >100e18)
    assertGt(inputAmount, 100e18); // High input due to low liquidity
    assertEq(poolToken.balanceOf(slippageVictim), uint256(10_000e18 -
        inputAmount)); // User loses more tokens than
         // expected
    vm.stopPrank();
}
```

Recommended Mitigation:

Add a `maxInputAmount` parameter and check to enforce slippage protection:

```
contract TSwapPool is ERC20 {
+   error TSwapPool__InputTooHigh(uint256 inputAmount, uint256
    maxInputAmount);

    function swapExactOutput(
        IERC20 inputToken,
        IERC20 outputToken,
        uint256 outputAmount,
+       uint256 maxInputAmount,
        uint64 deadline
-   ) public revertIfZero(outputAmount) revertIfDeadlinePassed(deadline
    ) returns (uint256 inputAmount) {
+   ) public revertIfZero(outputAmount) revertIfZero(maxInputAmount)
    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);
    }
}
```

LOW

[L-1] Unrestricted Pool Creation for Non-ERC20 Tokens like ERC721, etc (Improper Input Validation + Unexpected Behaviour)

Description:

The `PoolFactory::createPool` function allows pools to be created for any `tokenAddress`, without verifying if it is a valid ERC20 token. This enables pool creation for non-ERC20 contracts, such as ERC721 NFTs, which can lead to failures in `TSwapPool` operations due to missing ERC20 functions.

Impact:

Creating pools for ERC721 NFTs causes `TSwapPool` operations (e.g., deposit, swap) to fail due to the absence of ERC20 functions like `ERC20::transfer` and `ERC20::balanceOf` accurate selectors.

Proof of Concept:

In `PoolFactory.sol`, `createPool` does not validate `tokenAddress` for ERC20 compliance:

```
function createPool(address tokenAddress) external returns (address) {
    if (s_pools[tokenAddress] != address(0)) {
        revert PoolFactory__PoolAlreadyExists(tokenAddress);
    }
    string memory liquidityTokenName = string.concat("T-Swap ", IERC20(
        tokenAddress).name());
    // ... pool creation logic
}
```

Calling `IERC20(tokenAddress).name()` assumes ERC20 compliance, but if `tokenAddress` is an ERC721 contract, this call will succeed (since ERC721 has name()), yet other ERC20 functions like `transfer` will fail. Create a new contract named `ERC721Mock` in directory `test/mocks/`:

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.20;

import { ERC721 } from "@openzeppelin/contracts/token/ERC721/ERC721.sol
    ";

contract ERC721Mock is ERC721 {
    constructor(string memory name, string memory symbol) ERC721(name,
        symbol) { }

    function mint(address to, uint256 tokenId) external {
        _mint(to, tokenId);
    }

    function burn(uint256 tokenId) external {
        _burn(tokenId);
    }
}
```

Appending the following test in `PoolFactoryTest.t.sol` demonstrates the issue:

```
import { ERC721Mock } from "../mocks/ERC721Mock.sol";

function testPoolCreationWithERC721Token() public {
    // Deploy an ERC721 contract
    ERC721Mock nft = new ERC721Mock("TestNFT", "NFT");
    // Attempt to create pool with ERC721 token
    address poolAddress = factory.createPool(address(nft));
    assertEq(factory.getPool(address(nft)), poolAddress);
    // Pool creation succeeds with ERC721 token, but TSwapPool
        operations will fail
}
```

Recommended Mitigation:

Add validation in `PoolFactory::createPool` to ensure `tokenAddress` supports ERC20 functions, explicitly checking for `balanceOf` and `transfer` to distinguish from ERC721 and others:

```
contract PoolFactory {
+   error PoolFactory__InvalidERC20Token();

    function createPool(address tokenAddress) external returns (address
        ) {
+       if (tokenAddress == address(0)) {
+           revert PoolFactory__InvalidERC20Token();
+       }
+       // Check for ERC20 compliance by verifying key ERC20 functions
+       try IERC20(tokenAddress).balanceOf(address(this)) returns (
    uint256) {
+       } catch {
+           revert PoolFactory__InvalidERC20Token();
+       }
+       try IERC20(tokenAddress).transfer(address(this), 0) returns (
    bool) {
+       } catch {
+           revert PoolFactory__InvalidERC20Token();
+       }
        if (s_pools[tokenAddress] != address(0)) {
            revert PoolFactory__PoolAlreadyExists(tokenAddress);
        }
        // ... pool creation logic
    }
}
```

[L-2] Incorrect Parameter Order in TSwapPool::LiquidityAdded Event (Event Mismatch + Incorrect Offchain Accounting)

Description:

The TSwapPool::_addLiquidityMintAndTransfer function emits the LiquidityAdded event with incorrect parameter ordering. The event logs poolTokensToDeposit as the second parameter and wethToDeposit as the third, which mismatches the expected order (WETH first, pool tokens second), potentially causing incorrect accounting in application's front end.

Impact:

Incorrect event data will lead to errors in off-chain analytics and accounting systems tracking liquidity contributions whcih will in turn confuse users or integrators by misinformation, reducing trust and usability of the protocol.

Proof of Concept:

In TSwapPool.sol, the LiquidityAdded event is emitted with swapped parameters:

```
function _addLiquidityMintAndTransfer(
    uint256 wethToDeposit,
    uint256 poolTokensToDeposit,
    uint256 liquidityTokensToMint
) private {
    _mint(msg.sender, liquidityTokensToMint);
    emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit)
        ;
    // ... transfer logic
}
```

The event signature expects wethDeposited first, then poolTokensDeposited:

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

Appending this test in TSwapPool.t.sol demonstrates the issue:

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

function testLiquidityAddedEventWrongParameterOrder() public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 5e18); // 5 WETH
    poolToken.approve(address(pool), 100e18); // 100 pool tokens
    vm.expectEmit(true, false, false, true);
    // Expecting wrong emit // Expected: msg.sender, pool tokens, WETH
    emit LiquidityAdded(liquidityProvider, 100e18, 5e18);
    //Correct emit: emit LiquidityAdded(liquidityProvider, 5e18, 100e18
        ); msg.sender, WETH, pool tokens
    pool.deposit(5e18, 5e18, 100e18, uint64(block.timestamp));
    // Event emitted with poolTokensToDeposit (100e18) as second param,
         wethToDeposit (100e18) as third
    vm.stopPrank();
}
```

Recommended Mitigation:

Correct the parameter order in the `LiquidityAdded` event emission:

```
contract TSwapPool is ERC20 {
    function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
    ) private {
        _mint(msg.sender, liquidityTokensToMint);
-       emit LiquidityAdded(msg.sender, poolTokensToDeposit,
    wethToDeposit);
+       emit LiquidityAdded(msg.sender, wethToDeposit,
    poolTokensToDeposit);
        // ... transfer logic
    }
}
```

[L-3] Unused Return Parameter in `TSwapPool::swapExactInput` (Incorrect Return Value + Misleading Function Behavior)

Description:

The `TSwapPool::swapExactInput` function declares a return parameter output of type uint256, but it is not assigned a value within the function. As a result, it returns `0` by default, which is misleading and incorrect, as it should return the actual `outputAmount` calculated during the swap.

Impact:

Calling contracts expecting the output return value from `swapExactInput` receive `0`, leading to incorrect assumptions about the swap's outcome. This can cause errors in smart contracts that rely on the return value for subsequent logic, such as tracking token amounts or updating internal state, potentially leading to failed transactions or financial miscalculations, reducing the protocol's reliability and usability.

Proof of Concept:

In `TSwapPool.sol`, the `swapExactInput` function does not assign a value to `output`:

```
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
) public revertIfZero(inputAmount) revertIfDeadlinePassed(deadline)
   returns (uint256 output) {
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));
    uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
        inputReserves, outputReserves);
    if (outputAmount < minOutputAmount) {
        revert TSwapPool__OutputTooLow(outputAmount, minOutputAmount);
    }
    _swap(inputToken, inputAmount, outputToken, outputAmount);
    // output is never assigned, returns 0
}
```

Appending this test in `TSwapPool.t.sol` demonstrates the issue with a calling contract:

Proof Of Code:

```
function testSwapExactInputMisleadsCallingContract() public {
    // Deploy a calling contract
    CallerContract caller = new CallerContract(address(pool));

    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);
    weth.mint(address(caller), 10e18);
    // Caller contract expects non-zero output but gets 0
    vm.expectRevert("Swap output is zero");
    caller.performSwap(weth, 10e18, poolToken, 7e18, uint64(block.
        timestamp));
    vm.stopPrank();
}

// Test contract that calls swapExactInput
contract CallerContract {
    TSwapPool pool;

    constructor(address _pool) {
        pool = TSwapPool(_pool);
    }

    function performSwap(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        external
    {
        inputToken.approve(address(pool), inputAmount);
        uint256 output = pool.swapExactInput(inputToken, inputAmount,
            outputToken, minOutputAmount, deadline);
        require(output > 0, "Swap output is zero");
        // Further logic would use output, but its zero
    }
}
```

Recommended Mitigation:

Assign the calculated `outputAmount` to the output return parameter:

```
contract TSwapPool is ERC20 {
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    ) public revertIfZero(inputAmount) revertIfDeadlinePassed(deadline)
        returns (uint256 output) {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));
        uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount,
            inputReserves, outputReserves);
        if (outputAmount < minOutputAmount) {
            revert TSwapPool__OutputTooLow(outputAmount,
                minOutputAmount);
        }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
+           output = outputAmount;
    }
}
```