Protocol Audit Report

Version 1.0

Kevin Lee

August 27, 2025

Protocol Audit Report

Kevin Lee

August 26, 2025

Lead Auditors: Kevin Lee

Table of Contents

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

Kevin Lee 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.

Audit Details

Scope

```
src/PoolFactory.sol
src/TSwapPool.sol
```

Issue Summary

Severity	Count	Percentage
High	3	23%
Medium	3	23%
Low	2	15%
Informational	5	39%
Total	**13**	**100%**

High

[H-1] The `TSwapPool::_swap` will reward user when the swapping times up to 10. It will break the protocol invariant – AMM formula `x * y = constant(k)`

Description: The `_swap` function in the `TSwapPool` contract includes a reward mechanism that grants users a bonus for every 10 swaps they perform. This design choice creates a potential exploit where users could manipulate the system to receive excessive rewards.

Impact: If exploited, this could lead to a significant imbalance in the liquidity pool, violating the core AMM principle of maintaining a constant product formula. This could result in financial losses for the protocol and its users.

Proof of Concept:

Invariant.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import {Test} from "forge-std/Test.sol";
import {StdInvariant} from "forge-std/StdInvariant.sol";
import {PoolFactory} from "../../src/PoolFactory.sol";
import {TSwapPool} from "../../src/TSwapPool.sol";
import {Handler} from "./Handler.t.sol";
import {MyTokenMock} from "../mocks/MyTokenMock.sol";
import {WETHMock} from "../mocks/WETHMock.sol";

contract InvariantTest is StdInvariant, Test {
    MyTokenMock myToken;
    WETHMock wETH;
    address owner = makeAddr("owner");
    PoolFactory poolFactory;
    TSwapPool tswapPool;
    address tPool;
    uint256 constant STARTING_X = 1000 ether;
    uint256 constant STARTING_Y = 10 ether;
    Handler handler;

    // 1 WETH = 1000 MyToken
    uint256 constant K = 1000;

    function setUp() public {
        myToken = new MyTokenMock();
        myToken.mint(owner, STARTING_X);

        wETH = new WETHMock();
        wETH.mint(owner, STARTING_Y);

        poolFactory = new PoolFactory(address(wETH));
        tswapPool = TSwapPool(poolFactory.createPool(address(myToken)))
            ;

        handler = new Handler(tswapPool);

        // first Deposit
        vm.startPrank(owner);
        myToken.approve(address(tswapPool), type(uint256).max);
        wETH.approve(address(tswapPool), type(uint256).max);
        tswapPool.deposit(STARTING_Y, STARTING_Y, STARTING_X, uint64(
            block.timestamp));
        vm.stopPrank();

        targetContract(address(handler));
    }

    function invariant_AMMFormula() public view {
        assertEq(handler.EXPECTED_DELTA_X(), handler.ACTUAL_DELTA_X());
        assertEq(handler.EXPECTED_DELTA_Y(), handler.ACTUAL_DELTA_Y());
    }
}
```

Handler.t.sol

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

import {Test, console2} from "forge-std/Test.sol";
import {PoolFactory} from "../../src/PoolFactory.sol";
import {TSwapPool} from "../../src/TSwapPool.sol";
import {Handler} from "./Handler.t.sol";
import {MyTokenMock} from "../mocks/MyTokenMock.sol";
import {WETHMock} from "../mocks/WETHMock.sol";

contract Handler is Test {
    WETHMock wETH;
    MyTokenMock myToken;
    TSwapPool tswapPool;

    uint256 public STARTING_X;
    uint256 public STARTING_Y;
    uint256 public ENDING_X;
    uint256 public ENDING_Y;
    int256 public EXPECTED_DELTA_Y;
    int256 public EXPECTED_DELTA_X;
    int256 public ACTUAL_DELTA_Y;
    int256 public ACTUAL_DELTA_X;

    address swaper = makeAddr("swaper");
    address liquidityProvider = makeAddr("liquidityProvider");

    constructor(TSwapPool _tswapPool){
        wETH = WETHMock(_tswapPool.getWeth());
        myToken = MyTokenMock(_tswapPool.getPoolToken());
        tswapPool = TSwapPool(_tswapPool);
    }

    function deposit(uint256 _amountETH) public {
        uint256 amountETH = bound(
            _amountETH,
            tswapPool.getMinimumWethDepositAmount(),
            type(uint64).max
        );
        uint256 amountPoolToken = tswapPool.getPoolTokensToDepositBasedOnWeth(amountETH);

        // expected
        STARTING_X = wETH.balanceOf(address(tswapPool));
        STARTING_Y = myToken.balanceOf(address(tswapPool));
        EXPECTED_DELTA_X = int256(amountETH);
        EXPECTED_DELTA_Y = int256(amountPoolToken);

        // deposit
        vm.startPrank(liquidityProvider);
        if(wETH.balanceOf(address(liquidityProvider)) < amountETH) {
            wETH.mint(address(liquidityProvider), amountETH - wETH.balanceOf(address(liquidityProvider)));
        }
        if(myToken.balanceOf(address(liquidityProvider)) < amountPoolToken) {
            myToken.mint(address(liquidityProvider), amountPoolToken - myToken.balanceOf(address(liquidityProvider)));
        }
        wETH.approve(address(tswapPool), type(uint256).max);
        myToken.approve(address(tswapPool), type(uint256).max);
        tswapPool.deposit(amountETH,0,amountPoolToken,uint64(block.timestamp));
        vm.stopPrank();

        // actual
        ENDING_X = wETH.balanceOf(address(tswapPool));
        ENDING_Y = myToken.balanceOf(address(tswapPool));
        ACTUAL_DELTA_X = int256(ENDING_X) - int256(STARTING_X);
        ACTUAL_DELTA_Y = int256(ENDING_Y) - int256(STARTING_Y);
    }

    function swapPoolTokenOnOutputWETH(uint256 _amountWETH) public {
        uint256 amountETH = bound(_amountWETH,tswapPool.getMinimumWethDepositAmount(),uint256(type(uint64).max));

        if(amountETH > WETHMock(wETH).balanceOf(address(tswapPool))){
            return;
        }

        uint256 amountPoolToken = tswapPool.getInputAmountBasedOnOutput(
            amountETH,
            myToken.balanceOf(address(tswapPool)),
            wETH.balanceOf(address(tswapPool))
        );

        if(amountPoolToken > type(uint64).max){
            return;
        }

        //expect
        EXPECTED_DELTA_X = int256(-1) * int256(amountETH);
        EXPECTED_DELTA_Y = int256(amountPoolToken);
        STARTING_X = wETH.balanceOf(address(tswapPool));
        STARTING_Y = myToken.balanceOf(address(tswapPool));
        uint256 userBeforeWETH = wETH.balanceOf(address(swaper));

        // swap
        vm.startPrank(swaper);
        if(myToken.balanceOf(address(swaper)) < amountPoolToken) {
            myToken.mint(address(swaper), amountPoolToken - myToken.balanceOf(address(swaper)));
        }
        myToken.approve(address(tswapPool), type(uint256).max);
        tswapPool.swapExactOutput(myToken,wETH,amountETH,uint64(block.timestamp));
        vm.stopPrank();

        // actual
        ENDING_X = wETH.balanceOf(address(tswapPool));
        ENDING_Y = myToken.balanceOf(address(tswapPool));
        uint256 userAfterWETH = wETH.balanceOf(address(swaper));
        console2.log(userAfterWETH - userBeforeWETH);
        ACTUAL_DELTA_X = int256(ENDING_X) - int256(STARTING_X);
        ACTUAL_DELTA_Y = int256(ENDING_Y) - int256(STARTING_Y);
    }

}
```

Firstly, you should create a `Invariant` folder in `test` and add the `Invariant.t.sol`, `Handler.t.sol` to the folder. In `foundry.toml`, you should keep `fail_on_revert = true`.

Recommended Mitigation:

```
    /*//////////////////////////////////////////////////////////////
                            STATE VARIABLES
    //////////////////////////////////////////////////////////////*/
    IERC20 private immutable i_wethToken;
    IERC20 private immutable i_poolToken;
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
    uint256 private swap_count = 0;
-    uint256 private constant SWAP_COUNT_MAX = 10;
```

```
    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);
    }
```

Remove the reward mechanism to maintain the stability of the AMM.

[H-2] The TSwapPool::swapExactOutput function lacks slippage protection. As a result, users may end up spending more of one asset than intended to receive the desired amount of the other asset, potentially causing additional loss.

Description: The swapExactOutput function allows users to specify the exact amount of output tokens they want to receive. However, there is no mechanism to limit the maximum input amount a user is willing to pay. If the market price moves unfavorably during the transaction or due to front-running, the user may have to provide significantly more input tokens than expected.

Impact: Users may incur higher costs when swapping assets, leading to potential financial losses.

Proof of Concept:

```
    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
        );

        // @audit-high lack of slippage protection, might cause user lose assets

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

Recommended Mitigation:

```
error TSwapPool__DeadlineHasPassed(uint64 deadline);
    error TSwapPool__MaxPoolTokenDepositTooHigh(
        uint256 maximumPoolTokensToDeposit,
        uint256 poolTokensToDeposit
    );
    error TSwapPool__MinLiquidityTokensToMintTooLow(
        uint256 minimumLiquidityTokensToMint,
        uint256 liquidityTokensToMint
    );
    error TSwapPool__WethDepositAmountTooLow(
        uint256 minimumWethDeposit,
        uint256 wethToDeposit
    );
    error TSwapPool__InvalidToken();
    error TSwapPool__OutputTooLow(uint256 actual, uint256 min);
+   error TSwapPool__InputAmountTooHigh(uint256 actual, uint256 max);
    error TSwapPool__MustBeMoreThanZero();
```

```
    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__InputAmountTooHigh(inputAmount, maxInputAmount);
+       }
        _swap(inputToken, inputAmount, outputToken, outputAmount);
    }
```

[H-3] Incorrect precision in `TSwapPool::getInputAmountBasedOnOutput`. The function currently uses a precision of `10000`, but according to the project documentation, it should use `1000`. Each swap charges a `0.3%` fee, represented in both `getInputAmountBasedOnOutput` and `getOutputAmountBasedOnInput` by applying a `997/1000` multiplier. This fee is retained within the protocol.

Description: The function `getInputAmountBasedOnOutput` is responsible for calculating the input amount required to obtain a specific output amount in a swap. The current implementation uses a precision of `10000`, which is inconsistent with the project's documentation.

Impact: Using the incorrect precision can lead to inaccurate calculations of input amounts, potentially resulting in users receiving less output than expected or being charged higher fees.

Proof of Concept:

```
function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        return
            // @audit-info Use constant Variable to replace magic number
            // @audit-high the precision is 1000 instead of 10000
            ((inputReserves * outputAmount) * 10000) /
            ((outputReserves - outputAmount) * 997);
    }
```

Recommended Mitigation:

```
    /*//////////////////////////////////////////////////////////////
                            STATE VARIABLES
    //////////////////////////////////////////////////////////////*/
    IERC20 private immutable i_wethToken;
    IERC20 private immutable i_poolToken;
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
    uint256 private swap_count = 0;
    uint256 private constant SWAP_COUNT_MAX = 10;
+    uint256 private constant FEE_PRECISON = 1000;
+    uint256 private constant FEE_NUMERATOR = 997;
```

```
function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        return
            // @audit-info Use constant Variable to replace magic number
            // @audit-high the precision is 1000 instead of 10000
-            ((inputReserves * outputAmount) * 10000) /
-            ((outputReserves - outputAmount) * 997);
+            ((inputReserves * outputAmount) * FEE_PRECISON) /
+            ((outputReserves - outputAmount) * FEE_NUMERATOR);
    }
```

Add the correct constant variable to replace the wrong magic number

Medium

[M-1] In `TSwapPool::deposit`, the `deadline` variable is not used, it may cause MEV.

Description: The `deadline` variable is intended to prevent transactions from being processed after a certain time, but if it is not used in the function logic, it could allow for front-running or other forms of MEV (Miner Extractable Value) attacks.

Impact: If an attacker is able to front-run a transaction, they could potentially profit at the expense of the user, leading to financial losses.

Recommended Mitigation:

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
+        revertIfDeadlinePassed(deadline)
        returns (uint256 liquidityTokensToMint)
        .............. other code
```

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

[M-2] `TSwapPool::sellPoolTokens` function used the `TSwapPool::swapExactOutput` but it should use `TSwapPool::swapExactInput`

Description: The `sellPoolTokens` function is intended to sell a specific amount of pool tokens for a desired output amount of another token. However, it currently uses the `swapExactOutput` function, which is not suitable for this use case.

Impact: Using `swapExactOutput` instead of `swapExactInput` can lead to unexpected behavior and potential loss of funds, as the two functions have different mechanisms for handling token swaps.

Proof of Concept:

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

Recommended Mitigation:

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

+        swapExactInput(
+                i_poolToken,
+                poolTokenAmount,
+                i_wethToken,
+                0,    // Slippage protection, it just asumed closed
+                uint64(block.timestamp)
+            );
    }
```

Use the `swapExactInput` replaced the `swapExactOutput`.

[M-3] In `PoolFactory::createPool`, a user can create a malicious ERC20 contract that overrides the name function. Since `s_tokens[address(tPool)] = tokenAddress` is updated only after calling name, this can lead to a reentrancy attack. And the `liquidityTokenSymbol` used the name instead of `symbol`.

Description: When `PoolFactory::createPool` is called, the factory contract queries the ERC20 token's `name()` function before updating the internal mapping `s_tokens[address(tPool)] = tokenAddress`. If the provided token is a malicious ERC20 implementation, it can override the `name()` function to include arbitrary logic, such as calling back into the `PoolFactory::createPool` function. Since the mapping update occurs after the external call, the attacker can re-enter and repeatedly create multiple pools for the same token or manipulate the factory's internal state in unexpected ways.The `PoolFactory::liquidityTokenSymbol` that uses the `name` will return the wrong symbol.

Impact: Attacker might create the same pool to disrupted the system running, but `IERC20` limited the `name` is a staticall it will prevent many attacks.

Proof of Concept:

MaliciousERC20.sol

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

import { PoolFactory } from "../../src/PoolFactory.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {Context} from "@openzeppelin/contracts/utils/Context.sol";
import {IERC20Errors} from "@openzeppelin/contracts/interfaces/draft-IERC6093.sol";

contract MaliciousERC20 is Context, IERC20, IERC20Errors {
    mapping(address account => uint256) private _balances;

    mapping(address account => mapping(address spender => uint256)) private _allowances;

    uint256 private _totalSupply;

    bool private hasReentered = false;

    string private _name;
    string private _symbol;

    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    // malicious name()
    function name() public virtual returns (string memory) {
        if(hasReentered) {
            return _name;
        }
        hasReentered = true;
        address victim = msg.sender;
        PoolFactory pf = PoolFactory(victim);
        pf.createPool(address(this));
        return _name;
    }

    function symbol() public view virtual returns (string memory) {
        return _symbol;
    }
    function decimals() public view virtual returns (uint8) {
        return 18;
    }
    function totalSupply() public view virtual returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view virtual returns (uint256) {
        return _balances[account];
    }
    function transfer(address to, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
```

```solidity
        _transfer(owner, to, value);
        return true;
    }

    function allowance(address owner, address spender) public view virtual returns (uint256) {
        return _allowances[owner][spender];
    }
    function approve(address spender, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, value);
        return true;
    }
    function transferFrom(address from, address to, uint256 value) public virtual returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, value);
        _transfer(from, to, value);
        return true;
    }

    function _transfer(address from, address to, uint256 value) internal {
        if (from == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        if (to == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(from, to, value);
    }

    function _update(address from, address to, uint256 value) internal virtual {
        if (from == address(0)) {
            _totalSupply += value;
        } else {
            uint256 fromBalance = _balances[from];
            if (fromBalance < value) {
                revert ERC20InsufficientBalance(from, fromBalance, value);
            }
            unchecked {
                _balances[from] = fromBalance - value;
            }
        }

        if (to == address(0)) {
            unchecked {
                _totalSupply -= value;
            }
        } else {
            unchecked {
                _balances[to] += value;
            }
        }

        emit Transfer(from, to, value);
    }

    function _mint(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
    }

    function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), value);
    }

    function _approve(address owner, address spender, uint256 value) internal {
        _approve(owner, spender, value, true);
    }

    function _approve(address owner, address spender, uint256 value, bool emitEvent) internal virtual {
        if (owner == address(0)) {
            revert ERC20InvalidApprover(address(0));
        }
        if (spender == address(0)) {
            revert ERC20InvalidSpender(address(0));
        }
        _allowances[owner][spender] = value;
        if (emitEvent) {
            emit Approval(owner, spender, value);
        }
    }

    function _spendAllowance(address owner, address spender, uint256 value) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance < type(uint256).max) {
            if (currentAllowance < value) {
                revert ERC20InsufficientAllowance(spender, currentAllowance, value);
            }
            unchecked {
                _approve(owner, spender, currentAllowance - value, false);
            }
        }
    }
}
```

Test function

```
function testReenterancyAttackByName() public {
    MaliciousERC20 token = new MaliciousERC20("Malicious Token", "MAL");
    vm.expectRevert(
        abi.encodeWithSelector(
            PoolFactory.PoolFactory__PoolAlreadyExists.selector,
            address(tokenA)
        )
    );
    factory.createPool(address(token));
}
```

Create the `MaliciousERC20.t.sol` in uint folder and put test function in `PoolFactoryTest.t.sol`

Recommended Mitigation:

```
    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());
-        string memory liquidityTokenSymbol = string.concat("ts", IERC20(tokenAddress).name());
-       TSwapPool tPool = new TSwapPool(tokenAddress, i_wethToken, liquidityTokenName, liquidityTokenSymbol);
        s_pools[tokenAddress] = address(tPool);
        s_tokens[address(tPool)] = tokenAddress;
+        string memory liquidityTokenName = string.concat("T-Swap ", IERC20(tokenAddress).name());
+        string memory liquidityTokenSymbol = string.concat("ts", IERC20(tokenAddress).symbol());
+        TSwapPool tPool = new TSwapPool(tokenAddress, i_wethToken, liquidityTokenName, liquidityTokenSymbol);
        emit PoolCreated(tokenAddress, address(tPool));
        return address(tPool);
    }
```

Low

[L-1] In `TSwapPool::_addLiquidityMintAndTransfer`, the parameters sequence are wrong and event should be `LiquidityAdded(msg.sender, wethToDeposit, poolTokensToDeposit)` instead of `LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit)`

Description: The parameters for the `LiquidityAdded` event are in the wrong order, which can lead to confusion.

Impact: User may feel confused when they see the event.

Recommended Mitigation:

```
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);

        // Interactions
        i_wethToken.safeTransferFrom(msg.sender, address(this), wethToDeposit);
        i_poolToken.safeTransferFrom(
            msg.sender,
            address(this),
            poolTokensToDeposit
        );
    }
```

[L-2] The function `TSwapPool::swapExactInput` has a wrong return `output`. It always return 0

Description: The function `swapExactInput` is supposed to return the output amount of tokens after a swap, but it always returns 0.

Impact: This can lead to confusion for users who expect a non-zero output amount after a swap.

Proof of Concept:

```
function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        // @audit-low protocol is always return 0. It looks like is outputAmount
        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 a unused variable **Recommended Mitigation:**

```
function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        // @audit-low protocol is always return 0. It looks like is outputAmount
-        returns (uint256 output)
+        returns (uint256 outputAmount)
    {
        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);
    }
```

Return the `outputAmount`.

informational

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` is not used

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

[I-2] `PoolFactory::constructer` lack of zero check

```
  constructor(address wethToken) {
+   if(wethToken == address(0)){
+       revert "costum error"
    }
    i_wethToken = wethToken;
  }
```

[I-3] In `TSwapPool::deposit`, the constant variable suggest not to emmit in event

Recommended Mitigation:

```
error TSwapPool__MaxPoolTokenDepositTooHigh(
-    uint256 maximumPoolTokensToDeposit,
    uint256 poolTokensToDeposit
  );
```

```
if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
      revert TSwapPool__WethDepositAmountTooLow(
-                 MINIMUM_WETH_LIQUIDITY,
                wethToDeposit
            );
        }
```

Remove the `MINIMUM_WETH_LIQUIDITY` from the revert statement.

[I-4] In `TSwapPool::deposit`, the ’liquidityTokensToMint`update before the`_addLiquidityMintAndTransfer‘ is better.

Recommended Mitigation:

```
+   liquidityTokensToMint = wethToDeposit;
    _addLiquidityMintAndTransfer(
        wethToDeposit,
        maximumPoolTokensToDeposit,
        wethToDeposit
    );
-    liquidityTokensToMint = wethToDeposit;
```

[I-5] Use constant variable to replace the magic number.

Recommended Mitigation:

```
    /*//////////////////////////////////////////////////////////////
                            STATE VARIABLES
    //////////////////////////////////////////////////////////////*/
    IERC20 private immutable i_wethToken;
    IERC20 private immutable i_poolToken;
    uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000;
    uint256 private swap_count = 0;
    uint256 private constant SWAP_COUNT_MAX = 10;
+    uint256 private constant FEE_PRECISON = 1000;
+    uint256 private constant FEE_NUMERATOR = 997;
```

getOutputAmountBasedOnInput

```
function getOutputAmountBasedOnInput(
        uint256 inputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(inputAmount)
        revertIfZero(outputReserves)
        returns (uint256 outputAmount)
    {
        uint256 inputAmountMinusFee = inputAmount * FEE_NUMERATOR;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * FEE_PRECISON) +
           inputAmountMinusFee;
        return numerator / denominator;
    }
```

getInputAmountBasedOnOutput

```
function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
        return
            ((inputReserves * outputAmount) * FEE_PRECISON) /
            ((outputReserves - outputAmount) * FEE_NUMERATOR);
    }
```