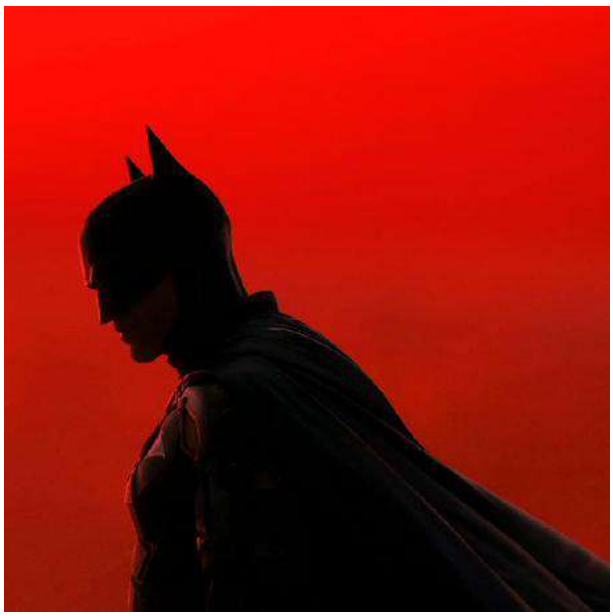

Protocol Audit Report

Version 1.0

Cyfrin.io

September 6, 2025

TSwapPool Audit Report

Cyfrin.io

March 7, 2023

Prepared by: Cyfrin Lead Auditors:

- Alqasem Hasan

Table of Contents

- Table of Contents
- Protocol Summary
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees
    * [H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens
    * [H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens.

    * [H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of
- Medium
    * [M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline
    * [M-2] Rebase, fee-on-transfer, and ERC777 tokens break protocol invairant.
- Lows
    * [L-1] `TSwapPool::LiquidityAdded` event has parameters out of order
    * [L-2] The `TSwapPool::swapExactInput` function always returns 0 incorrectly
- Informationals
    * [I-1] The error `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed
    * [I-2] The PoolFactory doesn't have a zero check for the wethToken address
    * [I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`
    * [I-4] Event is missing `indexed` fields
    * [I-5] The `TSwapPool` constructor doesn't have zero address checks for weth and pool tokens
    * [I-6] The `TSwapPool::deposit` function makes an internal call before changing a variable
    * [I-7] The `TSwapPool::swapExactInput` function is public when it should be external
- Gas
    * [G-1] `TSwapPool::MINIMUM_WETH_LIQUIDITY` is a constant and does not add utility when emitted in an event
    * [G-2] The `TSwapPool::deposit` function has an unused variable local variable that wastes gas

Protocol Summary

This protocol is an Automated Market Maker protocol like uniswap, but in only supports pools that swap WETH for any other ERC20 token. The protocol also allows for the creating of these pools via the PoolFactory contract.

Disclaimer

The Alqasem 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

Scope

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	4
Medium	2

Severity	Number of issues found
Low	2
Info	7
Gas	2
Total	17

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes protocol to take too many tokens from users, resulting in lost fees

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user should deposit given an amount of output tokens. However, the function currently miscalculates the resulting amount. When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Recommended Mitigation: Consider making the following change to the function.

```
function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
+        return ((inputReserves * outputAmount) * 1_000) / ((
    outputReserves - outputAmount) * 997);
-        return ((inputReserves * outputAmount) * 10_000) / ((
    outputReserves - outputAmount) * 997);
    }
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

Description: The `swapExactOutput` function does not include any sort of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput`,where the function specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact: If market conditions change before the transaction processes, the user could get a much worse swap.

Proof of Concept:

1. The price of weth right now is 1,000 USDC
2. User inputs a `swapExactOutput` looking for 1 WETH
    1. inputToken = USDC
    2. outputToken = Weth
    3. outputAmount = 1
    4. deadline = whatever
3. The function does not offer a maxInput amount
4. As the transaction is pending in the mempool, the market changes! And the price change is HUGE -> 1 WETH is now 10,000 USDC. 10x more than the user expected.
5. The transaction completes, but the user sent the protocol 10,000 USDC instead of the expected 1,000 USDC

Proof of Code:

```
1
2  function testSlippage() {
3          uint256 initialLiquidity = 100e18;
4          vm.startPrank(liquidityProvider);
5          weth.approve(address(pool), 100e18);
6          poolToken.approve(address(pool), 100e18);
7
8          // Deposit liquidity into the pool via a liquidity provider
9
10         pool.deposit({
11             wethToDeposit: initialLiquidity,
12             minimumLiquidityTokensToMint: 0,
13             maximumPoolTokensToDeposit: initialLiquidity,
14             deadline: uint64(block.timestamp)
15         });
16         vm.stopPrank();
17
18         // We initilize a user with 11 pool tokens
19
```

```
20          address someUser = makeAddr("someUser");
21          uint256 userInitialPoolTokenBalance = 11e18;
22          poolToken.mint(someUser, userInitialPoolTokenBalance);
23
24          // Initilize another user with 100_000 pool tokens
25          address richUser = makeAddr("richUser");
26          uint256 userInitialPoolTokenBalance = 100000e18;
27          poolToken.mint(richUser, userInitialPoolTokenBalance);
28          vm.startPrank(richUser);
29
30          // We get the price of WETH in poolTokens
31
32          uint256 originalWethPrice = pool.getPriceOfOneWethInPoolTokens
                ();
33          console.log("The original weth price is: ");
34          console.log(originalWethPrice);
35
36          // User 1 wants to buy 1 WETH from the pool, paying with pool
                tokens expecting the original weth price, but then a whale
                purchases a lot of WETH
37          // The whale decreases the supply of WETH, which increases the
                price of Weth, and User 1 has to pay the higher price.
38
39          // Rich user / Whales purcahse of weth
40          poolToken.approve(address(pool), type(uint256).max);
41          pool.swapExactOutput(poolToken, weth, 10 ether, uint64(block.
                timestamp));
42          vm.stopPrank();
43
44          // Some users purcahse
45          vm.startPrank(richUser);
46
47          poolToken.approve(address(pool), type(uint256).max);
48          pool.swapExactOutput(poolToken, weth, 1 ether, uint64(block.
                timestamp));
49          vm.stopPrank();
50
51      }
```

Recommended Mitigation: We should include a `maxInputAmount` so the user guarentees he only can spend up to a certain limit, and they can predict how much they will spend on the protocol.

```
1       function swapExactOutput(
2           IERC20 inputToken,
3   +        uint256 maxInputAmount,
4   .
5   .
6   .
7       inputAmount = getInputAmountBasedOnOutput(outputAmount,
            inputReserves, outputReserves);
8   +   if(inputAmount > maxInputAmount){
```

```
9  +         revert();
10 +     }
11       _swap(inputToken, inputAmount, outputToken, outputAmount);
```

[H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to receive the incorrect amount of tokens.

Description: The `sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. Users indicate how many pool tokens they're willing to sell in the `poolTokenAmount` parameter. However, the funcion currently miscalcuates the swapped amount.

This is due to the fact that the `swapExactOutput` function is called, whereas the `swapExactInput` function is the one that should be called. Because users specify the exact amount of input tokens, not output.

Impact: Users will swap the wrong amount of tokens, which is a severe disruption of protocol functionality.

Recommended Mitigation:

Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (ie `minWethToReceive` to be passed to `swapExactInput`)

```
1      function sellPoolTokens(
2          uint256 poolTokenAmount,
3  +       uint256 minWethToReceive
4      ) external returns (uint256 wethAmount) {
5  -         return swapExactOutput(i_poolToken, i_wethToken,
       poolTokenAmount, uint64(block.timestamp));
6  +         return swapExactInput(i_poolToken, poolTokenAmount,
       i_wethToken, minWethToReceive , uint64(block.timestamp));
7      }
```

Additionally, it might be wise to add a deadline to the function, as there is currently no deadline. (MEV later)

[H-4] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of

`X * Y = K`

Description: The protocol follows a strict invariant of `X * Y = K`. Where:

- x: The balance of the pool token
- y: The balance of WETH
- k: The constant product of the two balances

This means, that whenver the balances change in the protocol, the ratio betweem the two amounts should remain constant, hence the k. However, this is broken due to the extra incentive in the _swap function. Meaning that over time the protocol funds will be drained.

The follow block of code is responsible for the issue.

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

Impact: A user could malicously drain the protocol of all its funds by repeatedly swapping tokens until the protocol is empty.

Most simply put, the protocols core invariant is broken.

Proof of Concept:

1. A user swaps 10 times, and collects the extra incentive of 1_000_000_000_000_000_000 tokens.
2. That user continues to swap until all the protocol funds are drained.

Proof of Code

Place the following into TSwapPool.t.sol.

```
function testInvariantBroken() public {
        // Deposit liquidity into the pool via a liquidity provider

        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        // A user address that will try to break the invairant

        uint256 outputWeth = 1e17;

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp));
```

```
17          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
18          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
19          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
20          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
21          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
22          pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
                timestamp));
23
24          int256 startingY = int256(weth.balanceOf(address(pool)));
25          int256 expectedDeltaY = int256(-1) * int256(outputWeth);
26
27          vm.stopPrank();
28
29          uint256 endingY = weth.balanceOf(address(pool));
30          int256 actualDeltaY = int256(endingY) - startingY;
31          assertEq(actualDeltaY, expectedDeltaY);
32      }
```

Recommended Mitigation: Remove the extra incentive. If you want to keep this in, we should account for the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

```
1 -           swap_count++;
2 -           if (swap_count >= SWAP_COUNT_MAX) {
3 -               swap_count = 0;
4 -               outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
5 -           }
```

Medium

[M-1] `TSwapPool::deposit` is missing deadline check causing transactions to complete even after the deadline

Description: The `deposit` function accepts a deadline parameter, which according to the documentation is the deadline for the transaction to be completed by. However, this parameter is never used. As a consequence, operations that add liquidity to the pool might be executed at unexpected times, in market conditions where the deposit rate is unfavorable.

Impact: Transactions could be sent when market conditions are unfavorable to deposit, even when adding a deadline parameter.

Proof of Concept: The `deadline` parameter is unused.

Recommended Mitigation: Consider making the following change to the function.

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

[M-2] Rebase, fee-on-transfer, and ERC777 tokens break protocol invairant.

Description: Rebase, fee-on-transfer, and ERC777 tokens all break the invariant `X * Y = K`. Rebase tokens change balances without transfers. For example, if there is 100 WETH, and 100 Rebase tokens in a liquidity pool, `100 * 100 = 10_000`. So `k = 10_000`. But if a positive rebase happens, that increases the balance of the liquidity pool to 110 Rebase Tokens, then the new Constant Product would `100 * 110 = 11_000`, which changes the Constant Product, and the invariant is broken. Fee-on-transfer tokens reduce the amount received on transfer. For example, if a user sends 10 tokens to the pool, but there is a 10% fee, then the pool only receives 9 tokens, but it will calculate the swap as if it received 10 tokens, which breaks the invariant. ERC777 tokens have hooks/callbacks that can automatically call a function on transfer, and this function can make a call back into the pool, before it changes it's state, which allows the user to manipulat the pool state, and break the invairant.

Impact: All funds from the protocol can be drained by a malicious user.

Proof of Concept: Here is a proof of code attack for a ERC777 token.

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

import "@openzeppelin/contracts/token/ERC777/IERC777.sol";
import "@openzeppelin/contracts/token/ERC777/IERC777Sender.sol";
import "@openzeppelin/contracts/utils/introspection/IERC1820Registry.
    sol";

contract ERC777Attacker is IERC777Sender {
    IERC1820Registry constant registry = IERC1820Registry(
        0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24 // ERC1820 registry
    );

    IERC777 public token;
```

```
14      TSwapPool public amm;
15      bool internal reentered;
16
17      constructor(IERC777 _token, TSwapPool _amm) {
18          token = _token;
19          amm = _amm;
20
21          // register this contract as an ERC777 sender
22          registry.setInterfaceImplementer(
23              address(this),
24              keccak256("ERC777TokensSender"),
25              address(this)
26          );
27      }
28
29      // This hook is called before transferFrom() finalizes
30      function tokensToSend(
31          address, address, address, uint256, bytes calldata, bytes
                calldata
32      ) external override {
33          if (!reentered) {
34              reentered = true;
35
36              // Reenter AMM during the vulnerable transfer
37              amm.swapTokenForETH(1); // can call again with arbitrary
                    logic
38          }
39      }
40
41      function attack(uint256 amount) external {
42          // Start the attack by calling AMM swap
43          amm.swapTokenForETH(amount);
44      }
45  }
```

1. A user adds liquidity to the pool with a rebase token.
2. A positive rebase happens, increasing the balance of the rebase token in the pool
3. A user swaps tokens, and the invariant is broken, allowing the user to drain all the funds.

Recommended Mitigation:

Lows

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran` function, it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the

third parameter position, whereas the `wethToDeposit` value should go second.

Impact: Event emission is incorrect, leading to off-chain functions potentially malfunctioning.

Recommended Mitigation:

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

[L-2] The `TSwapPool::swapExactInput` function always returns 0 incorrectly

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the names return value `output` it is never assigned a value, nor does it use an explicit return statement.

Impact: The return value will always be 0, giving incorrect info to the caller.

Proof of Concept: (or proof of code)

Recommended Mitigation:

Informationals

[I-1] The error `PoolFactory::PoolFactory__PoolDoesNotExist` is not used and should be removed

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

[I-2] The PoolFactory doesn't have a zero check for the wethToken address

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

[I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()`

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

[I-4] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

4 Found Instances

- Found in src/PoolFactory.sol Line: 35

```
- event PoolCreated(address tokenAddress, address poolAddress);
+ event PoolCreated(address indexed tokenAddress, address indexed poolAddress);
```

- Found in src/TSwapPool.sol Line: 43

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

- Found in src/TSwapPool.sol Line: 44

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

- Found in src/TSwapPool.sol Line: 45

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

[I-5] The `TSwapPool` constructor doesn't have zero address checks for weth and pool tokens

Recommended Mitigation: Consider making the following change to the function.

```
constructor(
        address poolToken,
        address wethToken,
        string memory liquidityTokenName,
        string memory liquidityTokenSymbol
    )
        ERC20(liquidityTokenName, liquidityTokenSymbol)
    {
+       if(poolToken == address(0) || wethToken == address(0)){
+           revert();
+}
s        i_wethToken = IERC20(wethToken);
        i_poolToken = IERC20(poolToken);
    }
```

[I-6] The `TSwapPool::deposit` function makes an internal call before changing a variable

Recommended Mitigation: Consider making the following change to the function.

```
else {
            // This will be the "initial" funding of the protocol. We
               are starting from blank here!
            // We just have them send the tokens in, and we mint
               liquidity tokens based on the weth
-           _addLiquidityMintAndTransfer(wethToDeposit,
    maximumPoolTokensToDeposit, wethToDeposit);
            liquidityTokensToMint = wethToDeposit;
+           _addLiquidityMintAndTransfer(wethToDeposit,
    maximumPoolTokensToDeposit, wethToDeposit);
        }
```

[I-7] The `TSwapPool::swapExactInput` function is public when it should be external

Description: The `TSwapPool::swapExactInput` function is public but is never used within the contract so it should be public.

Recommended Mitigation: Consider making the following change to the function.

```
function swapExactInput(
        IERC20 inputToken, // e input token to swap / sell ie: DAI
        uint256 inputAmount, // e  amount of the input token to swap
        IERC20 outputToken, // e the output token to buy / ie weth
```

```
5           uint256 minOutputAmount, // e minimum output amount to recieve
                of weth from dai
6           uint64 deadline // e deadline for when the transaction should
                expire
7       )
8   +       sexternal
9   -       public
10          revertIfZero(inputAmount)
11          revertIfDeadlinePassed(deadline)
12          returns (
```

Gas

[G-1] `TSwapPool::MINIMUM_WETH_LIQUIDITY` is a constant and does not add utility when emitted in an event

Recommended Mitigation: Consider making the following change to the error and the revert:

```
1   -           error TSwapPool__WethDepositAmountTooLow(uint256
        minimumWethDeposit, uint256 wethToDeposit);
2   +           error TSwapPool__WethDepositAmountTooLow(uint256
        wethToDeposit);
3
4   -           revert TSwapPool__WethDepositAmountTooLow(
        MINIMUM_WETH_LIQUIDITY, wethToDeposit);
5   +           revert TSwapPool__WethDepositAmountTooLow(wethToDeposit);
```

[G-2] The `TSwapPool::deposit` function has an unused variable local variable that wastes gas

Recommended Mitigation: Consider making the following change to the function.

```
1   if (totalLiquidityTokenSupply() > 0) {
2               uint256 wethReserves = i_wethToken.balanceOf(address(this))
                    ;
3   -           uint256 poolTokenReserves = i_poolToken.balanceOf(address(
        this));
```