TSwap Protocol Audit Report

Version 1.0

SpectraD

August 30, 2024

Protocol Audit Report

SpectraD

August 30, 2024

Prepared by: SpectraD Lead Auditors: - SpectraD

Table of Contents

Protocol Summary

This audit took approximately 10 hours to complete and proved to be particularly challenging. The use of artificial intelligence tools was very helpful in conducting this thorough analysis. The proofs of concept (PoCs) were especially complex to develop, requiring a deep understanding of the protocol's inner workings.

Disclaimer

The SpectraD 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
Low	2
Info	4
Total	12

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 tokens 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.

Proof of Concept:

```
    function testIncorrectFeeCalculation() public view {
        uint256 inputReserves = 1000 ether;
        uint256 outputReserves = 1000 ether;
        uint256 outputAmount = 10 ether;

        // Incorrect calculation (with getInputAmountBasedOnOutput())
        uint256 incorrectInputAmount = pool.getInputAmountBasedOnOutput
            (outputAmount, inputReserves, outputReserves);

        // Correct calculation
        uint256 correctInputAmount = (inputReserves * outputAmount *
            1000) / ((outputReserves - outputAmount) * 997);

        console.log("Good one:", incorrectInputAmount);
        console.log("Bad one:", correctInputAmount);

        assertGt(incorrectInputAmount, correctInputAmount, "Amount of
            input is too high");
    }
```

Recommended Mitigation:

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
```

```
11      {
12  -       return ((inputReserves * outputAmount) * 10_000) / ((
        outputReserves - outputAmount) * 997);
13  +       return ((inputReserves * outputAmount) * 1_000) / ((
        outputReserves - outputAmount) * 997);
14      }
```

Or just don't use magic numbers.

[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 transaciton processes, the user could get a much worse swap.

Proof of Concept: 1. The price of 1 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 moves 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

Exploit test

```
1     function testSwapExactOutputVulnerability() public {
2         // Initial setup
3         vm.startPrank(liquidityProvider);
4         weth.approve(address(pool), 100e18);
5         poolToken.approve(address(pool), 100e18);
6         pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
7         vm.stopPrank();
8
9         // Mint more tokens for the user
10        weth.mint(user, 1000e18);
11        poolToken.mint(user, 1000e18);
12
13        vm.startPrank(user);
14
15        // Approve a very high amount of input tokens
16        poolToken.approve(address(pool), type(uint256).max);
17
18        // Record initial balance of input token
19        uint256 initialBalance = poolToken.balanceOf(user);
```

```
20
21          // Desired output amount
22          uint256 outputAmount = 10e18;
23
24          // Perform the swap
25          uint256 inputAmount = pool.swapExactOutput(
26              poolToken,
27              weth,
28              outputAmount,
29              uint64(block.timestamp + 1 hours)
30          );
31
32          // Check how many input tokens were actually spent
33          uint256 actualInputAmount = initialBalance - poolToken.
                balanceOf(user);
34
35          // This assertion will pass, showing that the function works as
                expected in normal conditions
36          assertEq(actualInputAmount, inputAmount, "Input amount should
                match the calculated amount");
37
38          // Now, let's simulate a market change by adding more
                poolTokens to the pool
39          // This will make the exchange rate less favorable for the user
40          poolToken.mint(address(pool), 50e18);
41
42          // Perform another swap with the same output amount
43          uint256 newInputAmount = pool.swapExactOutput(
44              poolToken,
45              weth,
46              outputAmount,
47              uint64(block.timestamp + 1 hours)
48          );
49
50          // Check how many input tokens were actually spent this time
51          uint256 newActualInputAmount = poolToken.balanceOf(user);
52          newActualInputAmount = initialBalance - newActualInputAmount;
53
54          // This assertion will pass, showing that more input tokens
                were spent for the same output amount
55          assertGt(newActualInputAmount, actualInputAmount, "New input
                amount should be greater due to unfavorable market change");
56
57          // If there was a maxInputAmount parameter, the transaction
                would have reverted here
58          // Since there isn't, the user ends up spending more than they
                might have intended
59
60          console.log("Initial swap:");
61          console.log("Input amount:", inputAmount);
62          console.log("Actual input amount:", actualInputAmount);
```

```
63
64          console.log("\nAfter market change:");
65          console.log("New input amount:", newInputAmount);
66          console.log("New actual input amount:", newActualInputAmount);
67
68          console.log("\nDifference in input amounts:",
               newActualInputAmount - actualInputAmount);
69
70          vm.stopPrank();
71      }
```

Recommended Mitigation: We should include a `maxInputAmount` so the user only has to spend up to a specific amount, and 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 function currently miscalculaes 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 protcol functionality.

Proof of Concept:

Exploit test

```
1       function testSellPoolTokensIncorrectAmount() public {
```

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

        // User gets pool tokens
        vm.startPrank(user);
        poolToken.mint(user, 1000e18); // Mint more tokens to ensure sufficient balance
        poolToken.approve(address(pool), type(uint256).max); // Approve max amount

        // Record initial WETH balance
        uint256 initialWethBalance = weth.balanceOf(user);

        // Attempt to sell pool tokens
        uint256 poolTokensToSell = 5e18;
        uint256 expectedWethAmount = pool.getOutputAmountBasedOnInput(
            poolTokensToSell,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );

        uint256 receivedWethAmount = pool.sellPoolTokens(poolTokensToSell);

        // Log the results
        console.log("Pool tokens sold:", poolTokensToSell);
        console.log("Expected WETH amount:", expectedWethAmount);
        console.log("Received WETH amount:", receivedWethAmount);
        console.log(
            "Difference:",
            receivedWethAmount > expectedWethAmount
                ? receivedWethAmount - expectedWethAmount
                : expectedWethAmount - receivedWethAmount
        );

        // Verifications
        assertEq(
            receivedWethAmount,
            poolTokensToSell,
            "Received amount should equal the amount of pool tokens sold"
        );
        assertEq(
            weth.balanceOf(user) - initialWethBalance,
            poolTokensToSell,
            "WETH balance should increase by the amount of pool tokens sold"
```

```
48          );
49          assertNotEq(
50              receivedWethAmount,
51              expectedWethAmount,
52              "Received amount should not equal the expected amount"
53          );
54
55          vm.stopPrank();
56      }
```

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.

[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 whenever the balances change in the protocol, the ratio between 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.

```
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           }
```

Impact: A user could maliciously drain the protocol of funds by doing a lot of swaps and collecting the extra incentive given out by the protocol.

Most simply put, the protocol's 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 untill all the protocol funds are drained

Proof Of Code

Place the following into `TSwapPool.t.sol`.

```
1
2     function testInvariantBroken() public {
3         vm.startPrank(liquidityProvider);
4         weth.approve(address(pool), 100e18);
5         poolToken.approve(address(pool), 100e18);
6         pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
7         vm.stopPrank();
8
9         uint256 outputWeth = 1e17;
10
11        vm.startPrank(user);
12        poolToken.approve(address(pool), type(uint256).max);
13        poolToken.mint(user, 100e18);
14        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
              timestamp));
15        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
              timestamp));
16        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        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
              timestamp));
28        vm.stopPrank();
29
```

```
30          uint256 endingY = weth.balanceOf(address(pool));
31          int256 actualDeltaY = int256(endingY) - int256(startingY);
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

Recommended Mitigation: Remove the extra incentive mechanism. 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 -         // Fee-on-transfer
3 -         if (swap_count >= SWAP_COUNT_MAX) {
4 -             swap_count = 0;
5 -             outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
6 -         }
```

Findings

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, operationrs 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.

```
1 function deposit(
2         uint256 wethToDeposit,
3         uint256 minimumLiquidityTokensToMint, // LP tokens -> if empty,
              we can pick 100% (100% == 17 tokens)
4         uint256 maximumPoolTokensToDeposit,
5         uint64 deadline
6     )
7         external
8 +       revertIfDeadlinePassed(deadline)
```

```
9           revertIfZero(wethToDeposit)
10          returns (uint256 liquidityTokensToMint)
11      {
```

[M-2] Rebase, fee-on-transfer, and ERC-777 tokens break protocol invariant

Description: The `TSwapPool` contract has a critical vulnerability where the WETH balance of the pool increases unexpectedly after deposits. This breaks the core invariant of the protocol, which should maintain a constant product of token balances (x * y = k) after each operation.

In our invariant test, we observed that after a single deposit operation, the WETH balance of the pool increased from the expected 50e18 to approximately 63.6e18. This significant and unintended increase occurred without any swap operations, indicating a fundamental flaw in the deposit or internal accounting logic.

Impact: This vulnerability has severe implications:

1. Breaks Core Invariant: The unexpected increase in WETH balance violates the constant product formula (x * y = k), which is fundamental to the proper functioning of the AMM (Automated Market Maker).
2. Economic Exploitation: Attackers could potentially exploit this flaw to drain funds from the pool by strategically depositing and withdrawing tokens.
3. Incorrect Price Calculations: The inflated balance will lead to incorrect price calculations for swaps, potentially causing users to receive fewer tokens than they should.
4. Loss of User Funds: Liquidity providers may lose funds as the pool's total value increases artificially, diluting the value of LP tokens.
5. Protocol Instability: Over time, this issue could lead to significant imbalances in the pool, potentially rendering the entire protocol unusable.

Proof of Concept: The vulnerability was discovered through invariant testing. Here's a simplified version of the test that exposed the issue:

Exploit test

Invariants.t.sol

```
1   contract Invariants is StdInvariant, Test {
2       // Code
3       uint256 public constant SWAP_COUNT_MAX = 10;
4
5       function setUp() public {
6           // Code
```

```
            targetSelector(
                FuzzSelector({addr: address(handler), selectors:
                    selectors})
            );
            targetContract(address(handler));
        }
        //Code
    }
```

```
    function statefulFuzz_swapCounterInvariant() public view {
        // Verify that the swap count has not exceeded SWAP_COUNT_MAX
        assertTrue(handler.getSwapCount() <= SWAP_COUNT_MAX);

        // Verify that the WETH balance of the pool has not been
            modified unexpectedly
        uint256 expectedWethBalance = uint256(STARTING_Y) + handler.
            getTotalWethDeposited() - handler.getTotalWethWithdrawn();
        assertEq(weth.balanceOf(address(pool)), expectedWethBalance);
    }
```

Handler.t.sol

```
+    import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.
    sol";

    contract Handler is Test {
        // Code
        uint256 public totalWethDeposited;
        uint256 public totalWethWithdrawn;
        uint256 public swapCount;

        //Code

        function swapExactInput(uint256 inputAmount) public {
            // Be sure the handler has enough tokens for the swap
            deal(address(poolToken), address(this), inputAmount);
            poolToken.approve(address(pool), inputAmount);

            uint256 minOutputAmount = 1; // Minimum output amount to
                avoid errors
            pool.swapExactInput(IERC20(address(poolToken)), inputAmount
                , IERC20(address(weth)), minOutputAmount, uint64(block.
                timestamp));

            swapCount++;
            totalWethWithdrawn += minOutputAmount;
        }

        function getSwapCount() public view returns (uint256) {
            return swapCount;
        }
```

```
26
27          function getTotalWethDeposited() public view returns (uint256)
                {
28              return totalWethDeposited;
29          }
30
31          function getTotalWethWithdrawn() public view returns (uint256)
                {
32              return totalWethWithdrawn;
33          }
34      }
```

Recommended Mitigation:

1. Carefully review the `deposit` function in the `TSwapPool` contract. Ensure that it correctly calculates and transfers the exact amount of tokens without any unintended additions.
2. Implement a strict balance check immediately after the deposit operation:

```
1       function deposit(uint256 wethToDeposit, ...) external returns (
            uint256 liquidityTokensToMint) {
2           uint256 initialWethBalance = weth.balanceOf(address(this));
3           // ... existing deposit logic ...
4           uint256 finalWethBalance = weth.balanceOf(address(this));
5           require(finalWethBalance == initialWethBalance + wethToDeposit,
                "Unexpected balance change");
6           // ... rest of the function ...
7       }
```

3. Review any internal functions called during the deposit process, especially `_addLiquidityMintAndTransf`, to ensure they don't inadvertently add extra tokens.
4. Implement a global invariant check function that verifies the constant product formula (x * y = k) after every state-changing operation:

```
1       function checkInvariant() internal view {
2           uint256 x = poolToken.balanceOf(address(this));
3           uint256 y = weth.balanceOf(address(this));
4           uint256 k = x y;
5           require(k == lastKValue, "Invariant broken");
6       }
```

Call this function at the end of `deposit`, `withdraw`, and `swap` functions.

5. If there's any bonus or reward mechanism in the contract (e.g., for frequent swappers), ensure it's not accidentally triggered during deposits.
6. Increase the test coverage, particularly focusing on edge cases in deposit amounts and frequency.

7. Consider implementing a circuit breaker or emergency stop mechanism that can be triggered if significant balance discrepancies are detected.

After implementing these changes, re-run the invariant tests with an increased number of runs and various deposit scenarios to ensure the issue has been fully resolved.

Findings

Low

[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] Default value returned by `TSwapPool::swapExactInput` results in incorrect return value given

Description: The `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return value `ouput` it is never assigned a value, nor uses an explict return statement.

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

Proof of Concept:

Swap verification test

```
    function testSwapExactInputReturnsZero() public {
        // initial setup
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

```

```
        // swap preparation
        vm.startPrank(user);
        uint256 inputAmount = 10e18;
        poolToken.approve(address(pool), inputAmount);
        uint256 expectedOutputAmount = pool.getOutputAmountBasedOnInput(
            inputAmount,
            poolToken.balanceOf(address(pool)),
            weth.balanceOf(address(pool))
        );

        // swap execution
        uint256 returnedAmount = pool.swapExactInput(
            poolToken,
            inputAmount,
            weth,
            1, // minOutputAmount low to avoid revert
            uint64(block.timestamp)
        );

        // verification
        assertEq(returnedAmount, 0, "amount returned should be 0");

        uint256 receivedAmount = weth.balanceOf(user);
        uint256 expectedAmountWithBonus = expectedOutputAmount + 10e18;
            // add bonus of 10 tokens

        assertGt(receivedAmount, 0, "User should have received WETH");
        assertEq(receivedAmount, expectedAmountWithBonus, "amount
           received is not correct");

        vm.stopPrank();
    }
```

Recommended Mitigation:

```
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

-       uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount
    , inputReserves, outputReserves);
+       output = getOutputAmountBasedOnInput(inputAmount,
    inputReserves, outputReserves);

-       if (output < minOutputAmount) {
-           revert TSwapPool__OutputTooLow(outputAmount,
    minOutputAmount);
+       if (output < minOutputAmount) {
+           revert TSwapPool__OutputTooLow(outputAmount,
    minOutputAmount);
```

```
12          }
13
14 -         _swap(inputToken, inputAmount, outputToken, outputAmount);
15 +         _swap(inputToken, inputAmount, outputToken, output);
16      }
```

Findings

Info

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

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

[I-2] Lacking zero address checks

```
1      constructor(address wethToken) {
2 +        if(wethToken == address(0)) {
3 +            revert();
4 +         }
5          i_wethToken = wethToken;
6      }
```

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

```
1 -         string memory liquidityTokenSymbol = string.concat("ts",
      IERC20(tokenAddress).name());
2 +         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: 37

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

- Found in src/TSwapPool.sol Line: 52

```
1       event LiquidityAdded(
```

- Found in src/TSwapPool.sol Line: 57

```
1       event LiquidityRemoved(
```

- Found in src/TSwapPool.sol Line: 62

```
1       event Swap(
```