Protocol Audit Report

Prepared by: Cyfrin

Table of Contents

Protocol Summary

Protocol is meant to be a permissionless way for users to swap assets between each other at a fair price. You can think of T-Swap as a decentralized asset/token exchange (DEX).

Disclaimer

Bikalpa 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: 1ec3c30253423eb4199827f59cf564cc575b46db

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 16hr auditing this protocol and found total of 18 vulnerabilities. One of these vulnerality is highly dangerous because it has potential of breaking the protocol invariant.

Issues found

Severity	Number of Issue Found
High	5
Medium	0
Low	2
gas	3
Info	8
Total	18

Findings

Highs

[H-1] `TswapPool::deposit` is missing deadline check causing the transaction to complete even after deadline

Description: In the `TSwapPool::deposit` function, it accepts deadline parameters "@param deadline The deadline for the transaction to be completed by". However, The deadline check is no where checked causing the transaction of user who set deadline to 1 minute to take more time to complete even 1 hours. The user might get less `LP` tokens as a reward than expected due to loss in slippage.

Impact: The transaction can be easily frontrunned resulting to loss of slippage of user who want to deposit by gaining less lp tokens.

Proof of Concept: The `deadline` parameters is unused

Recommended Mitigation:

```
  function deposit(
         uint256 wethToDeposit,
         uint256 minimumLiquidityTokensToMint,
         uint256 maximumPoolTokensToDeposit,
         uint64 deadline
     )
```

```
        external
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
```

[H-2] Incorrect fee calculation at `TSwapPool::getInputAmountBasedOnOutput`, written 10000 instead of 1000.

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

Impact: Protocol takes more fees than expected from user.

Proof of code : Add this in your test and run it "forge test --mt "test_getInputAmountBasedOnOutput" -vvv

▶ Code

```
function test_getInputAmountBasedOnOutput() external {
    uint inputReserves = 1000 ;
    uint outputReserves = 2000 ;
    uint outputAmount = 100 ;

    uint expected = ((inputReserves * outputAmount) * 1000) /
((outputReserves - outputAmount) * 997) ;
    uint reality = ((inputReserves * outputAmount) * 10000) /
((outputReserves - outputAmount) * 997);

    assertEq(expected , reality  , "Calculated inout doesnt match expected
output") ;
}
```

You should see [FAIL: Calculated inout doesnt match expected output: 52 != 527] due to written 10_000 instead of 1_000 inside `TSwapPool::getInputAmountBasedOnOutput`.

Recommended Mitigation:

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

[H-3] No slippage and frontrunnung protection for user at `TSwapPool::swapExactOutput` resulting the frontrunner to exploit the transaction and

user receive fewer tokens.

Description: The `TSwapPool::swapExactOutput` doesnot include any sort of slippage protection. Just like in `TSwapPool::swapExactInput` where there is `minOutputAmount` the `TSwapPool::swapExactOutput` should also specify `maxInpurAmount`.

Impact: If market condition changes before the transaction processes, the user may get much worse swap.

Proof of Concept:

1. The price of 1 weth 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 include max input amount.
4. As the transaction is pending in the mempool, the market changes and the price now is now 1 WETH -> 10_000 USDC. 10x more than the user expected.
5. The transaction is completed but the user pays 10_000 instead of expected 1_000 to the protocol.

?????????

Recommended Mitigation: We should include a `maxInputAmount` so that the user only has to spend up to a specific amount, and predicit how much they will spend on the protocol.

▶ code

```diff
    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(maxInputAmount < inputAmount) revert

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

[H-4] `TSwapPool::sellPoolTokens` mismatches the order of params while calling `swapExactOutput`.

Description: the `sellPoolToken` function is intended to allow users to easily sell pooltoken and receive weth in exchange. User indicate how many poolToken they are willing to sell in the `poolAmount` parameter. However, the function inintentionally miscalculates the swapped amount.

This is due to the fact that `swapExactInput` should be called instead `swapExactOutput` because the users specify exact amount of input token instead of output.

Impact: User will swap wrong amount of token which is a severe disruption of the protocol.

Proof of Concept: Paste the following code in your unit testing.

▶ Details

```
 function test_H4_sellPoolTokens_shouldUseSwapExactInput() external {
  vm.startPrank(liquidityProvider) ;
  weth.approve(address(pool) , type(uint256).max) ;
  poolToken.approve(address(pool) , type(uint256).max) ;
  pool.deposit(100e18 , 100e18 , 100e18 , uint64(block.timestamp));
  vm.stopPrank() ;

  vm.startPrank(user) ;
  weth.approve(address(pool) , type(uint256).max) ;
  poolToken.approve(address(pool) , type(uint256).max) ;

  uint expected = pool.swapExactInput(poolToken , 2e18 , weth , 16e17,
uint64(block.timestamp)) ;
console.log("User weth balance:", weth.balanceOf(user));
console.log("User poolToken balance:", poolToken.balanceOf(user));
uint reality = pool.sellPoolTokens(2e18) ;
assertEq(expected , reality );
  vm.stopPrank() ;

}
```

You will see `ERC20InsufficientBalance` kind of error instead of sucessing this is because you are actually buying more pooltoken instead of selling it.

Recommended Mitigation: Consider changing the implementation of `swapExactInput` instead of `swapExactOutput`.

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

[H-5] In `TSwapPool::_swap` the extra tokens given to the users after every 10 `count` breaks the protocol invariant of `x * y = k`.

Description: The protocol follows a srict invariant of x * y = k. Where,

- `x` the balance of poolToken
- `y` the balance of wethToken
- `k` the constant product of two balances

This means, that whenever the balances changes in the protocol the ratio between the two amount should remain constant, hence the `k`. However, this is broken after every 10 swaps due to extra incentives in the `_swap` function. Meaning that overtime the protocol funds will be drained.

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

Most simply, putting the protocol core invariant be broken.

Proof of Concept: Paste the following code in your unit testing contract.

▶ Details

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

    uint256 outputWeth = 1e17;

    vm.startPrank(user);
    poolToken.approve(address(pool), type(uint256).max);
    poolToken.mint(user, 10e18);
    pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
    pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
```

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

    int256 startingY = int256(weth.balanceOf(address(pool))) ;
    int256 expectedDeltaY = int256(-1) * int256(outputWeth) ;

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

    int256 endingY = int256(weth.balanceOf(address(pool))) ;
        int256 actualDeltaY = endingY - startingY ;

        assertEq(expectedDeltaY , actualDeltaY) ;

}
```

A user swaps 10 times, and collects extra incentives of 1e18 tokens. The user can continue to swap until the contract funds are drained.

Recommended Mitigation: Remove the extra incentive mechanism. If you want to keep it, we should consider changeing the invariant of this protocol of x * y = k. Or we should set aside the token the same way we do with fees.

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

Lows

[L-1] The event function emitting is out of order at `TSwapPool::_addLiquidityMintAndTransfer`.

Description: The event in `_addLiquidityMintAndTransfer` is written backward resulting the protocol to give wrong information as emit. The `wethToDeposit` parameter should go in second param and `poolTokensToDeposit` should go in third prameters.

Impact: Could lead to off-chain function malfunctioning.

Recommended Mitigation:

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

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

[L-2] The `output` params in return is returning nothing or incorrect at `TSwapPool::swapExactInput`.

Description: The `TSwapPool::swapExactInput` is expected to return the actual amount of token bought by the caller. However, while it declares the name written value `output` it is never assigned the value nor uses an explicit return statement.

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

Proof of Concept: Paste the following function in your unit test code.

▶ Details

```
function test_SwapExactInputReturnMistake() external {
    vm.startPrank(liquidityProvider) ;
    poolToken.approve(address(pool) , type(uint256).max);
    weth.approve(address(pool) , type(uint256).max);
    pool.deposit(100e18 , 100e18, 100e18 , uint64(block.timestamp)) ;
    vm.stopPrank();

    vm.startPrank(user) ;
       poolToken.approve(address(pool) , type(uint256).max);
    weth.approve(address(pool) , type(uint256).max);

    uint result = pool.swapExactInput(poolToken , 2e18 , weth , 16e17 ,
uint64(block.timestamp)) ;

    assertEq(result , 0) ;
    vm.stopPrank();
}
```

The function will pass ecause it returned 0 instead of returning outputAmount.

Recommended Mitigation:

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

Gas

[G-1] `TSwapPool::deposit::poolTokenReserves` is not actually used. This can be removed.

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

[G-2] `TSwapPool::swapExactInput` should use external instead of public in order to reduce gas cost since this function is not being used anywhere else in the contract.

```
  function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
-        public
+ external
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        returns (uint256 output)
```

[G-3] `TSwapPool::totalLiquidityTokenSupply` should be external instead of public since it is not being used within the contract

```
- function totalLiquidityTokenSupply() public view returns (uint256) {
+ function totalLiquidityTokenSupply() external view returns (uint256) {
        return totalSupply();
    }
```

Informationals

[I-1] `PoolFactory::PoolFactory__PoolDoesNotExist` error is not written anywhere so it should be removed.

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

[I-2] `PoolFactory::constructor` and `TSwapPool::constructor` is lacking 0 address check.

▶ Code

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

[I-3] Incorrect symbol assign in `PoolFactory::createPool`.

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

[I-4] Large literal value is not preferred to read. Make it simple to understand by using e notation in `TSwapPool::MINIMUM_WETH_LIQUIDITY`

```
-     uint256 private constant MINIMUM_WETH_LIQUIDITY = 1_000_000_000 ;
+     uint256 private constant MINIMUM_WETH_LIQUIDITY = 1e9 ;
```

[I-5] `TSwapPool::deposit` doesn't use CEI method in the else part at last.

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

[I-6] Magic Numbers of 997 and 1000 spotted at `TSwapPool::getOutputAmountBasedOnInput` and `TSwapPool::getInputAmountBasedOnOutput`.

```
uint constant percentageAfterFee = 997 ;
uint constant percentage = 1000 ;
  uint256 inputAmountMinusFee = inputAmount * percentageAfterFee;
        uint256 numerator = inputAmountMinusFee * outputReserves;
        uint256 denominator = (inputReserves * percentage) +
inputAmountMinusFee;
        return numerator / denominator;
```

[I-7] Nat Spec missing in `TSwapPool::swapExactInput` & `TSwapPool::swapExactOutput`.

[I-8] Large literal value is not preferred to read in `TSwapPool::_swap`.

```diff
    if (swap_count >= SWAP_COUNT_MAX) {
             swap_count = 0;
-             outputToken.safeTransfer(msg.sender, 1_000_000_000_000_000_000);
+             outputToken.safeTransfer(msg.sender, 1e18);

        }
```