Protocol Audit Report

Version 1.0

ZenoraSec

July 29, 2024

Protocol Audit Report

ZenoraSec

July 29, 2024

Lead Security Researcher: ZenoraSec Jeremy Zenora

Table of Contents

    * [H-4] Lack of slippage protection in `TSwapPool::swapExactOutput`, causing potentionally significant losses for users that swap
    * [H-5] Rebase tokens, and other "unusual" tokens with fee on transfer break the invariant equation!
- Medium
    * [M-1] In `TSwapPool::deposit` parameter `deadline` is not used, causing the transactions to complete even after specifiec deadline
- Low
    * [L-1] Event `TSwapPool::LiquidityAdded` is emited with parameters in incorrent order
    * [L-2] Return value of `TSwapPool::swapExactInput` will always be `0` because of wrong return parameter
- Informational
    * [I-1] `PoolFactory__PoolDoesNotExist` does not get used in the code and should be removed
    * [I-2] Constructor in `PoolFactory` lacks a zero-check for the input address
    * [I-3] In `PoolFactory::createPool` when assigning `liquidityTokenSymbol` you incorrectly call `.name()` instead of `.symbol()`
    * [I-4] In `TSwapPool::deposit` you have unused variable `poolTokenReserves`, remove it if you do not plan to use it
    * [I-5] Very important function `TSwapPool::swapExactInput` does not have any natspec
    * [I-6] Function `TSwapPool::swapExactInput` should be external instead of public, because it is not called anywhere in the contract, to be more gas-efficient

Protocol Summary

Protocol handles strorage and retrieval of user's password. Protocol is meant for one single user. Only the owner of the contract (single user) can change and access the password.

Disclaimer

The ZenoraSec 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
Likelihood	High	H	H/M	M
	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

```
./src/
#-- PoolFactory.sol
#-- TSwapPool.sol
```

Executive Summary

We found a few major issues that contradicted the logic and original use of the protocol. We recommend to rethink the way the whole protocol works as suggested in the findings below. We used foundry framework with tools such as fuzzing, symbolic/formal verification, manual review.

Issues found

Severity	Number of issues found
High	5
Medium	1
Low	2

Severity	Number of issues found
Info	6
Total	14

Findings

High

[H-1] Typo in function `TSwapPool::getInputAmountBasedOnOutput` on line `((inputReserves * outputAmount) * 10000)`, number `10_000` causes the swap fee to be 91.3%! Significantly higher than the intented 0.3% swap fee

Description: There is an additional zero added to the number `10_000`, instead of `1_000`:

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

This makes the fee unbearable for the users.

Impact: The fee becomes `91.3`% instead of `0.3`%

Recommended Mitigation: Remove one `0` from the number, use constant variables to express this number, it is not good to use `magic numbers` in the codebase.

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

[H-2] In function `TSwapPool::_swap` after 10 swaps, protocol adds 1 additional token, this breaks the invariant `x * y = k`

Description: If you add additional 1 token to the user after 10 swaps, you break the math of the protocol as described here: - `x` - The Balance of the pool token - `y` - The Balance of the WETH - `k` - The constant product of the two balances

Therefore whenever the balances change in the protocol, constant `k` should be ideal ratio between them, however this breaks once you add extra 1 token in the `_swap` function. Over time the protocol funds will be drained.

```
// x * y = k
// numberOfWeth * numberOfPoolTokens = constant k
// k must not change during a transaction (invariant)
```

this is the malicious code block:

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

Impact: This will break the invarinat k and drain funds of the contract over time, malicious user could swap many times and get very large amount of tokens.

Proof of Concept: 1. User makes 10 swaps 2. User gets additional token and repeats this until funds are gone from the protocol. 3. protocol invariant k changes and breaks.

PoC

```
function testInvariantBreaks() public {
        //we add liquidity
        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();
        uint256 outputWeth = 1e17;
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        //Lets make 10 swaps
        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)
        );

        vm.stopPrank();

        uint256 endingY = weth.balanceOf(address(pool));
        int256 actualDeltaY = int256(endingY) - int256(startingY);
        assertEq(actualDeltaY, expectedDeltaY);
```

```
    }
```

Recommended Mitigation: Remove the functionality of sending 1 token to the user, or find a different way to reward your users, for example a `claim` function where users can claim reward if they did enough transactions.

[H-3] Function `TSwapPool::sellPoolTokens` mismatches parameters when calling the `TSwapPool::swapExactOutput`, resulting in users receiving incorrect amount of WETH.

Description: In function `sellPoolTokens` users try to sell their exact pool token amount as input for weth, then function `swapExactOutput` gets called where we take outputAmount as parameter, for this reason we should call `TSwapPool::swapExactInput` if we want to sell exact input amount.

Impact: Users sell incorrect amount of tokens, which is severe disruption of the protocol functionality.

Proof of Concept: 1. User has 100e18 poolTokens. 2. User tries to sell ONLY 5 poolTokens using current `sellPoolTokens`. 3. User ends up selling completely different poolToken amount.

Output from my PoC below:

```
Logs:
  BEFORE: poolTokens of user:    100
  BEFORE: weth balance of user:  100
  AFTER: poolTokens of user:     47
  AFTER: weth balance of user:   105
```

PoC

```
function testSellPoolTokensAmount() public {
        //we add liquidity
        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();

        console.log(
            "BEFORE: poolTokens of user:   ",
            poolToken.balanceOf(user) / 1e18
        );
        console.log(
            "BEFORE: weth balance of user: ",
            weth.balanceOf(user) / 1e18
        );
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);

        //We try to sell ONLY 5 pool tokens
        pool.sellPoolTokens(5e18);
        vm.stopPrank();

        console.log(
            "AFTER: poolTokens of user:    ",
            poolToken.balanceOf(user) / 1e18
        );
        console.log(
            "AFTER: weth balance of user:  ",
            weth.balanceOf(user) / 1e18
        );
    }
```

Recommended Mitigation: To fix this, simply call swapExactInput instead of swapExactOutput . And add additional parameter to be passed to your sellPoolTokens, e.g. uint256 minOutputAmount.

```
function sellPoolTokens(
-       uint256 poolTokenAmount
+       uint256 poolTokenAmount, uint256 minOutputAmount
    ) external returns (uint256 wethAmount) {
        return
-           swapExactOutput(
-               i_poolToken,
-               i_wethToken,
-               //@reported-high this is wrong, use swapexactinput
-               poolTokenAmount,
-               uint64(block.timestamp)

+           swapExactInput(
+               i_poolToken,
+               poolTokenAmount,
+               i_wethToken,
+               minOutputAmount,
+               uint64(block.timestamp)
            );
    }
```

[H-4] Lack of slippage protection in `TSwapPool::swapExactOutput`, causing potentionally significant losses for users that swap

Description: in swapExactOutput you do not have any checks for slippage. This function is similar to TSwapPool::swapExactInput where you do check if the outputAmount is less than minOutputAmount which prevents from slipapge attacks. You should specify maxInputAmount in the `swapExactOutput`.

Impact: If the price changes significantly during the transaction, the user can get very bad rate.

Proof of Concept: scenario not accounting for fees: 1. Price of 1 WETH is 1000 USDC 2. User swaps for 1 WETH 1. input token is USDC 2. output token is WETH 3. we want 1 weth amount 4. deadline = whatever 3. The function does not offer maxInputAmount (so we can end up paying whatever for the 1 weth) 4. the price changes, now 1 WETH costs 10,000 USDC and we still pay it. 5. the user spends 10,000 USDC for the 1 WETH, 10x more than they expected.

PoC Logs:

```
Price of 1 WETH ON START: 0.91 /  914145202967408518 THAT USER THINKS HE IS BUYING AT
Price of 1 WETH AFTER FRONTRUN:  3
Price of 1 WETH AFTER USER BOUGHT  11
```

PoC

```
function testSlippageOnSwapExactOutput() public {
        //we initialize the attacker's balance
        address attacker = makeAddr("attacker");
        poolToken.mint(attacker, 100e18);
        weth.mint(attacker, 100e18);

        //we add liquidity using independent wallet
        vm.startPrank(liquidityProvider);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.deposit(11e18, 11e18, 11e18, uint64(block.timestamp));
        vm.stopPrank();

        //checking price of 1 WETH when the user looks at the pool and decides to make a swap.
        uint256 priceOfOneWETHBefore = pool.getPriceOfOneWethInPoolTokens();

        vm.startPrank(attacker);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.swapExactInput(
            poolToken,
            10e18,
            weth,
            1e18,
            uint64(block.timestamp)
        );
        vm.stopPrank();

        uint256 priceAfterAttackerFrontran = pool
```

```
            .getPriceOfOneWethInPoolTokens();

        //user victim tries to swap too, does not know the price increased and cannot set parameters for his desired price.
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        weth.approve(address(pool), type(uint256).max);
        pool.swapExactOutput(poolToken, weth, 1e18, uint64(block.timestamp));
        vm.stopPrank();
        uint256 priceAfterUserBought = pool.getPriceOfOneWethInPoolTokens();
        console.log(
            "Price of 1 WETH ON START: 0.91 or ",
            priceOfOneWETHBefore,
            " Because of the mistake in fee calculation, which takes 91%, we get 0, raw price number is 91..."
        );
        console.log(
            "Price of 1 WETH AFTER FRONTRUN: ",
            priceAfterAttackerFrontran / 1e18
        );
        console.log(
            "Price of 1 WETH AFTER USER BOUGHT ",
            priceAfterUserBought / 1e18
        );
    }
```

Recommended Mitigation:

Add parameter `uint256 maxInputAmount` to function, and then revert if the `inputAmount` required to buy desired output is greater than `maxInputAmount`.

```
    function swapExactOutput(
        IERC20 inputToken,
+       uint256 maxInputAmount,
        IERC20 outputToken,
        uint256 outputAmount,
        uint64 deadline
.
.
.

        inputAmount = getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

+       if (inputAmount > maxInputAmount)
+          { revert; }
```

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

[H-5] Rebase tokens, and other “unusual” tokens with fee on transfer break the invariant equation!

Description: If the TSwapPool is exposed to ERC20’s with unusual functionality, like fees on transfers, it can severly disrupt its functioning. the constant `k` is not equal to what it should be anymore.

Impact: Protocol logic fails.

Proof of Concept:

add this import to your test suite

```
import {WEIRDERC} from "./mocks/ERC20WEIRDERC.sol";
```

Add this Contract to your “./mocks/” folder next to ERC20Mock.sol

WEIRDERC Contract

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

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

contract WEIRDERC is ERC20 {
    uint256 public constant INITIAL_SUPPLY = 1_000_000e18;
    address public immutable owner;
    uint256 public constant FEE = 27; // Fee percentage
    uint256 public constant PRECISION = 100;

    event TransferWithFee(address from, address to, uint256 value,
        uint256 fee);

    constructor() ERC20("WEIRDERC", "WEIRDERC") {
        owner = msg.sender;
        _mint(msg.sender, INITIAL_SUPPLY);
    }

    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }

    function transfer(
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        _transferWithFee(_msgSender(), to, amount);
        return true;
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, amount);
        _transferWithFee(from, to, amount);
        return true;
    }

    function _transferWithFee(
        address from,
        address to,
        uint256 amount
    ) internal {
        uint256 fee = (amount * FEE) / PRECISION;
        uint256 amountAfterFee = amount - fee;
        emit TransferWithFee(from, to, amountAfterFee, fee); // Emit event for debugging
        super._transfer(from, owner, fee); // Burn the fee by sending to zero address
        super._transfer(from, to, amountAfterFee);
    }
}
```

Fee on Transfer messes up our pool, now it actual delta is not the expected delta.

```
assertion failed: 5399025358904997822 != -10000000000000000000
```

PoC

```
function testWeirdERC20Behaviour() public {
        vm.startPrank(liquidityProvider);
        WEIRDERC weirdPoolToken = new WEIRDERC();
        TSwapPool weirdPool = new TSwapPool(
            address(weirdPoolToken),
            address(weth),
            "WeirdToken",
            "WEIRD"
        );

        weirdPoolToken.mint(liquidityProvider, 100e18);
        weirdPoolToken.approve(address(weirdPool), type(uint256).max);
        weth.approve(address(weirdPool), type(uint256).max);
        weirdPool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 outputWeth = 1e18;

        vm.startPrank(user);
        weirdPoolToken.mint(user, 100e18);
        weirdPoolToken.approve(address(weirdPool), type(uint256).max);
        weth.approve(address(weirdPool), type(uint256).max);

        int256 startingX = int256(weirdPoolToken.balanceOf(address(
           weirdPool)));
        int256 expectedDeltaX = int256(-1) * int256(outputWeth);

        weirdPool.swapExactOutput(
            weirdPoolToken,
            weth,
            outputWeth,
            uint64(block.timestamp)
        );

        uint256 endingX = weirdPoolToken.balanceOf(address(weirdPool));
        int256 actualDeltaX = int256(endingX) - int256(startingX);

        assertEq(actualDeltaX, expectedDeltaX);
        vm.stopPrank();
    }
```

Recommended Mitigation: Consider restricting the protocol for use of only certain types of ERC20s, not all of them.

Medium

[M-1] In `TSwapPool::deposit` parameter `deadline` is not used, causing the transactions to complete even after specifiec deadline

Description: You ask for input parameter `deadline` which according to the natspec is: “The deadline for the transaction to be completed by”. But you do not use it later on in the function to actually limit the transaction to cancel after certain period.

Impact: Transaction can go through at random time, potentially hurting the depositor.

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

Recommended Mitigation: Add modifier to prevent transaction to complete after deadline

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        //@audit-high, parameter `deadline` is not used, users can call
            deposit function and have their tx go through even after
            specified deadline. Disruption of functionality.
        uint64 deadline
    )
        external
+           revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
```

Low

[L-1] Event `TSwapPool::LiquidityAdded` is emited with parameters in incorrent order

Description: When emitting the event in the `TSwapPool::_addLiquidityMintAndTransfer` function, it logs values in wrong order. the `poolTokensToDeposit` variable should be on 3rd spot and `wethToDeposit` should be 2nd.

This is how the event is looks:

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

Impact: This leads to incorrect off-chain data.

Recommended Mitigation: Swap the two variables

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

[L-2] Return value of `TSwapPool::swapExactInput` will always be `0` because of wrong return parameter

Description: You return `uint256 output`, but there is no value assigned to that variable in the function.

Impact: Function `TSwapPool::swapExactInput` will confuse users and return wrong value `0`.

Proof of Concept:

1. User calls the `swapExactInput` function
2. Return value logged is always `0` even with successful transactions and different values.

```
Logs:
  Return value:  0
```

PoC

```
function testReturnValueOfswapExactInput() public {
        //we add liquidity
        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);

        uint256 returnValue = pool.swapExactInput(
            poolToken,
            1e18,
            weth,
            1e17,
            uint64(block.timestamp)
        );
        vm.stopPrank();
        console.log("Return value: ", returnValue);
    }
```

Recommended Mitigation: Either rename the return value `output` to `outputAmount` or rename the `outputAmount` to `output` in the rest of the function as below.

```
function swapExactInput(IERC20 inputToken, uint256 inputAmount, IERC20
    outputToken, uint256 minOutputAmount, uint64 deadline) public
    revertIfZero(inputAmount)
        returns (
            //@audit-low returning 0 always
            uint256 output
        ) {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

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

-       if (outputAmount < minOutputAmount)
-           { revert TSwapPool__OutputTooLow(outputAmount,
    minOutputAmount); }
-       _swap(inputToken, inputAmount, outputToken, outputAmount);

+       if (output < minOutputAmount)
+          { revert TSwapPool__OutputTooLow(output, minOutputAmount); }
+       _swap(inputToken, inputAmount, outputToken, output);
    }
```

Informational

[I-1] `PoolFactory__PoolDoesNotExist` does not get used in the code and should be removed

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

[I-2] Constructor in `PoolFactory` lacks a zero-check for the input address

Consider adding a `require` statement to prevent against zero addresses

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

[I-3] In `PoolFactory::createPool` when assigning `liquidityTokenSymbol` you incorrectly call `.name()` instead of `.symbol()`

Change it to `.symbol()`

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

[I-4] In `TSwapPool::deposit` you have unused variable `poolTokenReserves`, remove it if you do not plan to use it

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

[I-5] Very important function `TSwapPool::swapExactInput` does not have any natspec

One of the two most important functions to swap tokens, does not have documentation, consider adding one like you do in `swapExactOutput`

```

    // NO DESCRIPTION
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
```

[I-6] Function `TSwapPool::swapExactInput` should be external instead of public, because it is not called anywhere in the contract, to be more gas-efficient

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