https://CryptoAudit.report `IONSwap` Smart Contract Security

Project Name: IONSwap

Audit Date: 2025.01.17

Document version: 1.1

Table of Contents

Introduction

Project Overview

The **IONSwap** smart contract is designed to facilitate token swaps between two ERC20 tokens at a fixed exchange rate, adjusted for their respective decimal representations. It supports both forward swaps (from `otherToken` to `pooledToken`) and reverse swaps (from `pooledToken` back to `otherToken`). The contract ensures fair and predictable exchanges by handling decimal adjustments and incorporates mechanisms for liquidity management.

Scope of Work

This security audit focuses exclusively on the `IONSwap` smart contract, analyzing its codebase to identify potential security vulnerabilities, logical errors, and adherence to best practices. The audit includes:

- Review of the implementation of token swapping functionality.
- Assessment of access controls and permission mechanisms.
- Evaluation of potential attack vectors, including reentrancy and integer overflows/underflows.
- Verification of compliance with Solidity coding standards and best practices.

Executive Summary

The `IONSwap` smart contract is well-structured and makes use of established OpenZeppelin libraries for security features such as reentrancy protection and safe mathematical operations. The contract implements necessary validations and error handling for token swaps and liquidity management.

Findings Overview:

- **Critical Severity:** None
- **High Severity:** None
- **Medium Severity:** None
- **Low Severity:** 2 issues identified
- **Informational:** 3 points of note

Overall, the contract is secure for deployment with consideration to the recommendations provided to address the low-severity and informational findings.

Findings

Critical Severity

No critical severity issues were found.

High Severity

No high severity issues were found.

Medium Severity

No medium severity issues were found.

Low Severity

L01: Potential Precision Loss in Token Exchange Calculations

Location:

- `getPooledAmountOut(uint256 _amount)`
- `getOtherAmountOut(uint256 _amount)`

Description:

The functions `getPooledAmountOut` and `getOtherAmountOut` perform calculations that involve division, which can lead to precision loss due to integer division in Solidity. This may result in users receiving less tokens than expected, especially for tokens with decimals other than 18.

Proof of Concept:

For example, when swapping a very small amount of tokens, the calculated `amountOut` might be zero due to integer division truncation:

```
amountOut = (_amount * pooledTokenRate) / otherTokenRate;
```

If `_amount * pooledTokenRate` is less than `otherTokenRate`, `amountOut` will be zero.

Impact:

Users might receive less tokens than expected, or the swap might revert due to `OutputAmountZero` error, leading to poor user experience.

Recommendation:

Consider implementing a minimum amount check and using a more precise calculation method, such as using a library that supports fixed-point arithmetic with rounding options to reduce precision loss.

Informational

I01: Use of `public` Visibility Instead of `external` for `getPooledAmountOut` and `getOtherAmountOut`

Location:

- `getPooledAmountOut(uint256 _amount)`
- `getOtherAmountOut(uint256 _amount)`

Description:

The functions are declared as `public`, but they are not called internally within the contract. Using `external` visibility can save gas when these functions are called externally.

Recommendation:

Change the function visibility from `public` to `external` to optimize gas usage.

I02: Missing Zero Address Check for `_receiver` in `withdrawLiquidity`

Location:

- `withdrawLiquidity(IERC20 _token, address _receiver, uint256 _amount)`

Description:

The `_receiver` address is not validated to ensure it is not the zero address. Transferring tokens to the zero address might lead to token loss.

Recommendation:

Add a check to ensure `_receiver` is not the zero address:

```
if (_receiver == address(0)) {
    revert InvalidReceiverAddress();
}
```

I03: Use of `revert` Statements Without Custom Error Messages in Constructor

Location:

- Constructor validations.

Description:

The constructor uses `revert` statements with custom errors. Ensure that these custom errors are properly defined and provide clear messages.

Recommendation:

Verify that all custom errors are adequately defined and consider adding more descriptive error messages if necessary.

Recommendations

1. **Address Precision Loss in Calculations:**

   Implement safe mathematical operations that handle decimal precision more accurately to prevent potential rounding errors that can affect token swap amounts.

2. **Validate Receiver Address in `withdrawLiquidity`:**

   Add a zero-address check for the `_receiver` parameter in the `withdrawLiquidity` function to prevent accidental token transfers to the zero address.

3. **Optimize Function Visibility:**

   Review the function visibility throughout the contract and use `external` for functions that are not called internally to save gas.

4. **Enhance Event Logging:**

   Ensure that all significant state changes, particularly ownership transfers, are adequately logged for transparency.

5. **Review Custom Errors and Messaging:**

   Ensure that all custom error messages provide clear and helpful information to assist in debugging and user experience.

Conclusion

The `IONSwap` smart contract is generally well-designed and implements essential security measures. The contract utilizes OpenZeppelin libraries for secure implementation of ownership, reentrancy guards, and safe token transfers.

While no critical vulnerabilities were found, addressing the low-severity issues and implementing the recommendations will enhance the contract's security and reliability. It is advisable to thoroughly test the contract with various token configurations and edge cases to ensure robust behavior under all conditions.
