https://CryptoAudit.report `IONBridgeRouter`

Smart Contract Security

Project Name: IONBridgeRouter

Audit Date: 2025.01.17

Document version: 1.1

Table of Contents

1. Introduction
2. Scope
3. Executive Summary
4. Critical Findings
5. Major Findings
6. Medium Findings
7. Low Findings
8. Informational Findings
9. Recommendations
10. Conclusion

Introduction

This report presents the results of the security audit performed on the **IONBridgeRouter** smart contract. The IONBridgeRouter acts as a facade for the `IONSwap` and `Bridge` contracts, providing a unified interface for users to perform cross-chain swaps and token transfers seamlessly between ICE v1 and ICE v2 tokens.

The primary goal of this audit is to ensure that the smart contract is secure, follows best practices, and is free from vulnerabilities that could compromise the integrity of the system or result in financial losses.

Scope

The scope of the audit includes the following smart contract:

- **IONBridgeRouter.sol**

Commit Hash: [The specific commit hash if available]

Files in Scope:

- `/contracts/ion-bridge-router/IONBridgeRouter.sol`

Out of Scope:

- External contracts and libraries such as `IONSwap`, `IIONBridge`, and OpenZeppelin contracts.

Executive Summary

The **IONBridgeRouter** smart contract has been analyzed for common security vulnerabilities, logical errors, and adherence to best practices. The contract is generally well-structured and implements safe token transfer

mechanisms using OpenZeppelin's `SafeERC20` library.

Summary of Findings:

- **Critical Severity**: 0
- **Major Severity**: 0
- **Medium Severity**: 1
- **Low Severity**: 2
- **Informational**: 3

Overall, the contract is secure; however, there are areas where improvements can be made to enhance security and robustness. The issues identified should be addressed before deployment to a production environment.

Critical Findings

No critical severity issues were found during the audit.

Major Findings

No major severity issues were found during the audit.

Medium Findings

M01: Missing Reentrancy Guards on Sensitive Functions

Location:

- `IONBridgeRouter.sol`: `burn()`, `voteForMinting()`

Description:

The `burn` and `voteForMinting` functions interact with external contracts and handle token transfers, approvals, and swaps. Without reentrancy guards, an attacker could exploit these functions by initiating reentrant calls, potentially causing unexpected behavior or draining funds.

Risk:

- **Reentrancy Attack**: An attacker could call these functions recursively before their initial execution completes, manipulating the state and causing financial loss.

Recommendation:

- Use OpenZeppelin's `ReentrancyGuard` contract and apply the `nonReentrant` modifier to functions that perform external calls and token transfers.
- Update the contract to inherit from `ReentrancyGuard`:

```
contract IONBridgeRouter is ReentrancyGuard {
    // ...
}
```

- Apply the `nonReentrant` modifier:

```
function burn(uint256 amount, IIONBridge.IONAddress memory addr) external
nonReentrant {
    // ...
}
```

Low Findings

L01: Lack of Input Validation for External Contract Addresses

Location:

- `IONBridgeRouter.sol` : Constructor

Description:

The constructor does not validate the addresses of the external contracts (`_iceV1` , `_iceV2` , `_bridge` , `_ionSwap`) passed during deployment. If any of these addresses are zero or invalid, it could lead to unexpected behavior or contract malfunction.

Risk:

- **Contract Misconfiguration**: Deploying the contract with invalid addresses could render it unusable or introduce vulnerabilities.

Recommendation:

- Add validation checks in the constructor to ensure that none of the addresses are zero:

```
constructor(
    address _iceV1,
    address _iceV2,
    address _bridge,
    address _ionSwap
) {
    require(_iceV1 != address(0), "Invalid ICE v1 token address");
    require(_iceV2 != address(0), "Invalid ICE v2 token address");
    require(_bridge != address(0), "Invalid Bridge contract address");
    require(_ionSwap != address(0), "Invalid IONSwap contract address");
    // ...
}
```

L02: Missing Events for Critical State-Changing Functions

Location:

- `IONBridgeRouter.sol` : `burn()` , `voteForMinting()`

Description:

The `burn` and `voteForMinting` functions perform critical operations, but they do not emit any events upon successful execution. The code contains TODO comments indicating the need to "Emit success events."

Risk:

- **Lack of Transparency**: Without events, it is difficult to track the execution of these functions, making auditing and monitoring challenging.

Recommendation:

- Implement events that log important information whenever `burn` and `voteForMinting` are called.

  Define events:

  ```
  event TokensBurned(address indexed user, uint256 amount,
  IIONBridge.IONAddress addr);
  event TokensMinted(address indexed user, uint256 iceV2Amount, uint256
  iceV1Amount);
  ```

  Emit events in the functions:

  ```
  function burn(uint256 amount, IIONBridge.IONAddress memory addr) external
  nonReentrant {
      // ...
      emit TokensBurned(msg.sender, iceV2Amount, addr);
  }

  function voteForMinting(IIONBridge.SwapData memory data,
  IIONBridge.Signature[] memory signatures) external nonReentrant {
      // ...
      emit TokensMinted(msg.sender, iceV2Amount, iceV1Amount);
  }
  ```

Informational Findings

I01: Use of Deprecated `approve` Function Pattern

Location:

- `IONBridgeRouter.sol`: `_swapIceV1ToV2()`, `_swapIceV2ToV1()`

Description:

The contract uses `approve` to set allowances before calling external contracts. This pattern can be susceptible to the known ERC20 approval race condition if not properly handled.

Recommendation:

- Use `safeIncreaseAllowance` or `safeApprove` from `SafeERC20` to manage allowances safely.
- Alternatively, set allowances to zero before setting a new value:

  ```
  iceV1.safeApprove(address(ionSwap), 0);
  iceV1.safeApprove(address(ionSwap), amount);
  ```

I02: Missing Error Messages on `require` Statements

Location:

- `IONBridgeRouter.sol`: `burn()`, `voteForMinting()`

Description:

The contract uses `require` statements with generic error messages. More descriptive error messages or custom errors can improve usability and debugging.

Recommendation:

- Define custom errors for better gas efficiency and clarity.

  Define custom errors:

  ```
  error InvalidAmount();
  error UnauthorizedReceiver();
  ```

  Use custom errors:

  ```
  require(amount > 0, "Amount must be greater than zero");
  // Replace with:
  if (amount == 0) revert InvalidAmount();
  ```

I03: Function Visibility Could Be Optimized

Location:

- `IONBridgeRouter.sol`: `_swapIceV1ToV2()`, `_swapIceV2ToV1()`

Description:

The `_swapIceV1ToV2` and `_swapIceV2ToV1` functions are marked as `internal` but are not called by any other functions inside the contract except one. Marking them as `private` could save gas by preventing the creation of unused function slots.

Recommendation:

- Change the visibility from `internal` to `private` if these functions are not intended to be inherited or used by derived contracts.

Recommendations

1. **Implement Reentrancy Guards**: Add reentrancy protection to functions that perform external calls and token transfers to prevent potential reentrancy attacks.
2. **Validate Constructor Inputs**: Ensure that all contract addresses provided during deployment are valid and not zero addresses.
3. **Emit Events**: Implement events for state-changing functions to improve transparency and facilitate off-chain monitoring and auditing.

4. **Use Safe Approve Patterns**: Adopt safe approval patterns to prevent potential issues with token allowances.
5. **Use Custom Errors**: Replace generic `require` statements with custom errors for better gas efficiency and clearer error reporting.
6. **Optimize Function Visibility**: Adjust function visibility to `private` where appropriate to optimize gas usage.

Conclusion

The **IONBridgeRouter** smart contract is generally well-designed and follows best practices in many areas. However, addressing the identified issues will enhance the security and robustness of the contract. Implementing the recommendations provided in this report is strongly advised before deploying the contract to a production environment.

By incorporating these changes, the contract will minimize potential vulnerabilities, ensure the safety of user funds, and improve maintainability and transparency.
