IARD SOLUTIONS

Protecting Web3

LIQLOCK SECURITY AUDIT

PUBLIC AUDIT

2023-10-02

audit@iard.solutions

https://iard.solutions

Introduction

IARD Solutions is at the forefront of the cryptocurrency industry, dedicated to the mission of decentralizing services and ensuring the highest standards of security and reliability. This report represents a comprehensive assessment conducted by our expert team at IARD Solutions, specializing in Smart Contract audits.

At IARD Solutions, we take pride in our rigorous and comprehensive audit approach. Our team of experts follows industry-best practices to ensure a thorough evaluation of Smart Contracts. Our audit methodology encompasses a combination of manual review and automated analysis tools, providing a multi-faceted assessment of the audited project's codebase. We adhere to the principles of transparency and accountability, employing a systematic and structured process to identify potential vulnerabilities, logic flaws, and security risks. By leveraging both static and dynamic analysis techniques, we assess the code's functionality, security, and adherence to industry standards.

The purpose of this report is to provide a thorough analysis of the LiqLock project's Smart Contract, assessing its integrity, functionality, and security. In this document, we present a concise overview of the audited project, outlining key findings and recommendations to enhance the project's robustness and trustworthiness in the decentralized landscape. Note that the audit's scope is limited to the source code provided to us at this time, referred in the audit scope section.

In the audit below, our findings are set in different categories depending on their impact on the Smart Contract:

- **CRITICAL**: Critical vulnerabilities pose an immediate and severe threat to the security, functionality, or integrity of the system. Exploiting these vulnerabilities could result in significant damage or loss.
- **HIGH**: High-severity vulnerabilities are serious but may not be as urgent as critical ones. They have the potential to cause significant harm if exploited and should be addressed promptly.
- **MEDIUM**: Medium-severity vulnerabilities are important but may not pose an immediate threat. They should be addressed in a timely manner to prevent potential security issues.
- **LOW**: Low-severity vulnerabilities are minor issues that have limited impact and are less likely to be exploited. They should still be addressed as part of a comprehensive security strategy.
- **INFORMATIONAL**: Informational findings provide non-critical, supplementary information that may be relevant for improving overall system understanding.

1. Audit Scope

The audit conducted by IARD Solutions for the LiqLock project was a comprehensive examination of its Smart Contract ecosystem. Our assessment covered various critical aspects, including but not limited to:

1. **Code Integrity:** We thoroughly reviewed the Smart Contract codebase to ensure it adheres to best coding practices, follows industry standards, and is free from vulnerabilities.
2. **Functionality:** We assessed the functionality of the Smart Contract to verify that it performs as intended, executes transactions correctly, and handles edge cases effectively.
3. **Security:** Security is a paramount concern throughout the audit. We scrutinized the code for potential vulnerabilities such as re-entrancy attacks, overflows, and underflows to name the most commons, ensuring that the Smart Contract is robust against malicious attempts.
4. **Gas Optimization:** In the context of blockchain, efficient gas usage is vital. We optimized the code for gas consumption, enhancing cost-effectiveness and overall performance.
5. **Documentation:** We also reviewed documentation to ensure that it provides clear and accurate guidance for users and developers, promoting ease of understanding and integration.

This comprehensive scope allowed us to provide a holistic evaluation of the LiqLock project's Smart Contract, identifying strengths and areas for improvement to enhance its overall reliability and security. We produced this audit using both automatic tools and manual analysis to reduce risks as much as possible.

The audit is limited to the code provided by LiqLock team. The audit was done on the commit 3b31e61239deb365cc01e929ffeacd387412ab81 from their git. The updated audit was then done on the commit 703fc743fb9858d71a58e136f0b59e89f81716c5.

It is essential to acknowledge that while IARD Solutions conducts thorough audits with the utmost diligence and expertise, our assessments may not uncover all potential vulnerabilities or risks. The primary purpose of our audit is to provide a comprehensive evaluation of the Smart Contract's codebase and security practices as of the audit date. Our findings and recommendations are based on the information available at the time of the assessment. The rapidly evolving nature of blockchain and cybersecurity means that new vulnerabilities can emerge, and project circumstances may change post-audit.

2. Audit Results

2.1 Contract Overview

In our audit of the Smart Contract for the LiqLock, we conducted a comprehensive examination of the codebase, which is written in Solidity. The Smart Contract was compiled using Solidity 0.8.19 and it relies on several libraries, including OpenZeppelin Ownable, Pausable, ReentrancyGuard and IERC721. The project is deployed on BNB Chain.

The key functions of this Smart Contract play a pivotal role in its functionality and security. Some of the most critical functions include:

- createLiquidityLock: This function is responsible for creating a Liquidity Lock. It is crucial for the user to be able to create the lock, by transferring the Liquidity NFT to the LiqLock Smart Contract.
- collectLiquidityFees: This function allows the lock owner or the designated tax collector to collect any available liquidity fees that have been generated by the pool.
- withdrawLiquidityLock: This function allows the lock owner to retrieve its lock if the lock period is over. It is essential for the owner to truly be the owner of the lock by being able to retrieve it.

These functions, along with the overall structure of the Smart Contract, form the backbone of the project's functionality and security. Our audit focused on ensuring their correctness, efficiency, and adherence to best practices.

2.2 Unit Testing Coverage

Unit test coverage is a crucial aspect of ensuring the reliability and functionality of the Smart Contract. During our audit, we evaluated the extent of unit test coverage to gauge the comprehensiveness of testing within the codebase. The unit tests are designed to validate individual components and functions of the Smart Contract, helping identify potential issues early in the development process. We assessed the percentage of code covered by unit tests, examining their effectiveness in ensuring code correctness and preventing regressions. Our findings regarding unit test coverage are integral to our overall assessment of the project's code quality and robustness.

The Unit Testing Coverage for this project is as follow:

File Name	% Statements	% Branch	% Functions	% Lines	Uncovered lines
LiqLockV1	100	85.71	100	100	

The unit tests provide complete coverage for all the contract statements in LiqLock locker, demonstrating a thorough test suite. Moreover, the tests cover 85.71% of its branches, indicating a high level of code coverage. Additionally, all the application's functions undergo testing, and every line of code is fully covered. Remaining uncovered branchs are different require() branch that are not tested.

2.3 Audit Findings

Our audit team found the following problems in the LiqLock Smart Contract:

- **CRITICAL**: 0 vulnerabilities
- **HIGH**: 0 vulnerabilities
- **MEDIUM**: 2 vulnerabilities
- **LOW**: 1 vulnerability
- **INFORMATIONAL**: 0 vulnerabilities

2.2.1 Reentrancy vulnerabilities: MEDIUM

Problem

In withdrawLiquidityLock the liquidity NFT is transferred before modifying the contract storage, eventually allowing for two withdrawals of a single NFT. This is mitigated by the fact that this is an NFT and that by definition cannot be withdrawn two times as only one NFT per lock exist.

```
// Transfer the NFT back to the owner
lock.nftManager.safeTransferFrom(address(this), msg.sender, lock.nftId);

// Clean up the lock data
delete LIQ_LOCK_LOCKS[lockId];

// Remove the lock from the owner's set of locks
bool lockDeletionSuccess = LIQ_LOCKS_BY_OWNER[lock.owner].remove(
    lockId
);
require(
    lockDeletionSuccess,
    "LiqLockV1: Failed to remove lock from owner's set of locks"
);

// Emit an event to signify NFT withdrawal
emit LiquidityNFTWithdrawn(msg.sender, lockId);
```

Recommendation

We recommend modifying the storage before transferring to avoid any potential issues.

Updates

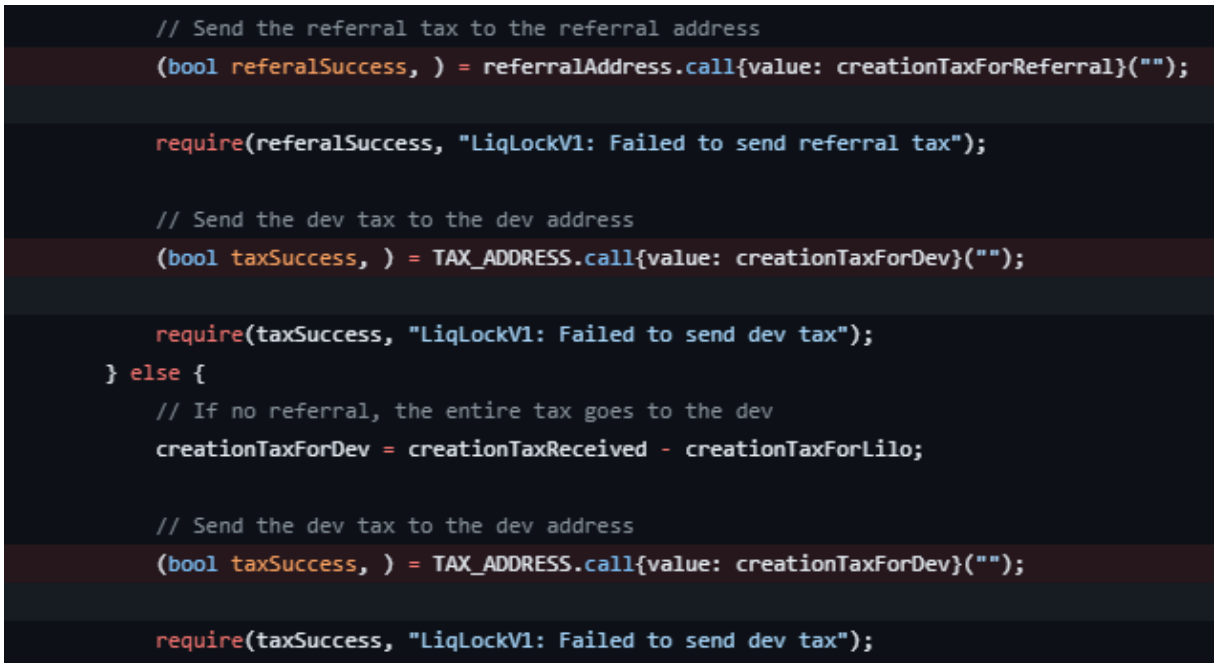
Acknowledged and fixed by LiqLock Team

```
// Clean up the lock data
delete LIQ_LOCK_LOCKS[lockId];

// Remove the lock from the owner's set of locks
bool lockDeletionSuccess = LIQ_LOCKS_BY_OWNER[lock.owner].remove(
    lockId
);
require(
    lockDeletionSuccess,
    "LiqLockV1: Failed to remove lock from owner's set of locks"
);

// Emit an event to signify NFT withdrawal
emit LiquidityNFTWithdrawn(msg.sender, lockId);

// Transfer the NFT back to the owner
lock.nftManager.safeTransferFrom(address(this), msg.sender, lock.nftId);
```

2.2.2 Unitialized local variable: MEDIUM

Problem

In createLiquidityLock success checking variables are not properly initialized with a value. This can lead to issues, while not in this specific case, and is highly recommended to fix at least as a good practice.



```
        // Send the referral tax to the referral address
        (bool referalSuccess, ) = referralAddress.call{value: creationTaxForReferral}("");

        require(referalSuccess, "LiqLockV1: Failed to send referral tax");

        // Send the dev tax to the dev address
        (bool taxSuccess, ) = TAX_ADDRESS.call{value: creationTaxForDev}("");

        require(taxSuccess, "LiqLockV1: Failed to send dev tax");
    } else {
        // If no referral, the entire tax goes to the dev
        creationTaxForDev = creationTaxReceived - creationTaxForLilo;

        // Send the dev tax to the dev address
        (bool taxSuccess, ) = TAX_ADDRESS.call{value: creationTaxForDev}("");

        require(taxSuccess, "LiqLockV1: Failed to send dev tax");
```

Recommendation

We recommend correctly initializing the different variables with a false value before using them.

Updates

Acknowledged and fixed by LiqLock Team

2.2.3 Local Variable Shadowing LOW

Problem

The variable owner is shadowed by the local variable owner in the getLickLockLocksByOwner function, which can cause issues. This is a LOW severity as it only impacts an external function and will not impact the actual functions of the contract.

Recommendation

Rename the local owner variable to another name, such as _lockOwner.

Updates

Acknowledged and fixed by LiqLock Team

3. Appendix