LeadWallet

Smart Contract Audit Report

IMMUNE BYTES

Audits

LEAD

March 12, 2021

Introduction

1. About LeadWallet

The Lead Wallet team is committed to providing a sophisticated yet simple crypto wallet application that will enable anyone (either newbie or expert) to store, send, receive, spend, exchange/swap crypto assets at users' convenience without the need to provide or store user data. Lead Wallet will enable users across the globe at any time to conveniently spend their cryptocurrency assets in exchange for what they've always wanted to have or buy. In addition, Lead Wallet will constantly research and provide excellent blockchain technology and cryptocurrency application scenarios that will further the adoption and use cases of cryptocurrencies.

Visit https://leadwallet.io/ to know more

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 15+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

LeadWallet team has provided documentation for the purpose of conducting the audit. The documents are:

1. Whitepaper
   https://www.leadwallet.io/pdf/Lead%20Wallet%20Whitepaper%201_0_2.pdf

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: LeadWallet
- Languages: Solidity(Smart contract)
- Github Link/Smart Contract Address for audit: https://etherscan.io/token/0x1dd80016e3d4ae146ee2ebb484e8edd92dacc4ce
- Deployed Smart Contract Address (kovan): https://kovan.etherscan.io/address/0x713394e7f9d044da8ce538fc06e57cb1ec0fab3f

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

Every issue in this report was assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	7
Closed	-	-	-

Low severity issues

1. **Wrong parameters used while emitting *OwnershipTransferred* event.**
   In **Owned** smart contract, an ***OwnershipTransferred*** event is defined which is emitted whenever the contract's ownership is changed. This event contains two parameters (previous and new owner's address). In **transferOwnership**() function the use of the event parameter is incorrect, it uses the new owner's address for both the variables. The statement at **Line 39** first updates the owner address, hence the **owner** and **_newOwner** variable refers to the same address.

   Recommendation:
   Consider emitting the event before the owner variable is updated.

2. ***Transfer* event not emitted in burn() and burnFrom() function.**
   In **LeadToken** smart contract, the **burn**() and **burnFrom**() functions are implemented by which users can burn their tokens, hence reduce the total supply of tokens. However, no **Transfer** event is emitted in both functions. The ERC20 standard **Transfer** event is monitored by various off-chain applications (like Etherscan) to track the movement of funds and the total supply of tokens. As per standards, whenever an account's balance is changed a **Transfer** event should be emitted.

   Recommendation:
   Consider emitting the **Transfer** event whenever tokens are burned as
   emit Transfer(account, address(0), _value);

3. ***OwnershipTransferred* event not emitted in the constructor() of Owned contract.**
   In the **constructor()** of **Owned.sol** the ownership of contract is initialized, i.e., changed from **zero** address to the deployer of contract. However, no **OwnershipTransferred** event was emitted. It is advised to emit the event whenever the owner of the smart contract is changed.

   Recommendation:
   Consider emitting the **OwnershipTransferred** event in the constructor as
   emit OwnershipTransferred(address(0), msg.sender);

4. **Revert reasons not provided.**
   In **LeadToken**, **Owned** and **SafeMath** smart contracts there are multiple places where **require** statements are used to validate certain conditions. However, no revert reasons are provided in some instances. It is always recommended to provide revert reasons in all **require** statements.

   Recommendation:
   Consider providing revert reasons in all **require** statements.

5. **Standard ERC20 events are defined twice in the smart contract.**
   In **LeadToken.sol,** the interface **ERC20Interface** is defined which contains the **Transfer** and **Approval** events. The **LeadToken** contract which inherits **ERC20Interface** also contains the **Transfer** and **Approval** events (**Line 62 and 75**). Defining events again does not provide any benefits and hence is never recommended.

   Recommendation:
   Consider removing the redeclared events from the **LeadToken** contract.

6. **Redundant conditional checks present.**
   In **LeadToken.sol**, there are multiple places where specific conditions are checked multiple times. At **Line 95** and **96**, a balance check and a **uint** overflow conditions are checked, however, these conditions are also verified at **Line 97** and **98** with the use of the **SafeMath** library. Hence the same conditions are checked twice. Similar is the case at **Line 184**, anyone trying to burn an amount more than his balance will be automatically reverted at **Line 185**. Additional computations in Solidity contracts result in more gas usage by transactions.

   Recommendation:
   Consider removing the redundant **require** statements, they can be safely removed.

7. **Unused internal _*mint()* function.**
   Currently, the **LeadToken** contract contains an **internal _mint**() function which mints a certain amount of token to a certain address provided as inputs. However this function is not used anywhere inside the smart contract, as this function is declared as **internal**, it cannot be invoked externally by other contracts or dapps. Hence the function is unusable.

   Recommendation:
   Consider removing the unusable function from the smart contract.

Unit Test

No unit tests were provided by the LeadWallet team.

Recommendation:
Our team suggests that the developers should write more extensive test cases for the contracts.

Coverage Report

Coverage report cannot be generated without unit test cases.

Recommendation:
We recommend 100% line and branch coverage for unit test cases.

Concluding Remarks

While conducting the audits of the LeadWallet smart contract, it was observed that the contracts contain only Low severity issues.

Resolving the areas is up to LeadWallet's discretion. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the LeadWallet platform or its product neither this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes Pvt Ltd.