LeadWallet
Smart Contract Audit Report

September 25, 2020

IMMUNE BYTES

Introduction

- ## 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

- ## About ImmuneBytes

  ImmuneBytes is a security start-up to provide professional services in 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. LEADSTAKE.pdf
2. Whitepaper
   https://leadwallet.io/en/docs/White%20Paper%20v1_0_2.pdf

Audit Process & Methodology

ImmueBytes 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 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 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), Javascript(Unit Testing)
- Github commit hash for audit: 1099c116912516f60e4fd78bc5fba3148b5c8676

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 in the following three categories:

1. Security: Identifying security related issues within each contract and 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 need to be fixed.

Issues	High	Medium	Low
Open	2	8	10
Closed	-	-	-

High severity issues

1. **Denial of Service (DoS)**
   As the LeadStake smart contract stores the address of users/stakers in an array **stakeholders[]**, at some point in future this array may become too large to be iterated over within the ethereum block gas limit. The iteration over this array is done every time any user unstake their LEAD tokens and whenever admin updates the weeklyROI. So a situation can occur where every unstaking or updating weeklyROI transaction may get reverted due to block gas limit.

   Recommendation:
   Smart contract implementations should be done in a way that no array becomes too large. For a quick solution a mapping can be created which stores the array index of a user's address in **stakeholders[].**
   mapping(address => uint) public userIndex;

2. **Admin can drain out all Lead balance.**
   The function **adminWithdraw**() is capable of draining out all LEAD token balance from LeadStaker smart contract. This could be done intentionally or unintentionally, i.e. in the case where admin's private key gets compromised.

   Recommendation:
   Consider adding a max cap to the value that the admin can withdraw. Admin should not be able to pull out funds more than the amount of total taxes collected from users.

Medium severity issues

1. **No check for address inputs variables in ERC20 transfer.**
   The **transfer**() function in ERC20.sol does not check the validity of input addresses by the user. This can lead to unintentional situations, hence not recommended.

   Recommendation:
   Consider adding checks for zero address (0x00) for ***to*** address. A check for ***to*** address should not be equal to LEAD token contract's address can also be added.

2. **No check for referrer address.**
   In **registerAndStake**() function, **referrer** address is taken as input from the user, the validity of which is not checked in the function which can be misused. Any address which is passed as referrer address collects referral rewards.

Users can provide the address of LeadStake contract as referrer, or any fresh address which has never interacted with LeadStake contract. That address will accrue referral rewards but won't be able to claim them which is entirely an unhandled scenario.

Recommendation:
Consider adding a require statement which checks if the referrer address is an registered address in LeadStake contract.

3. **Too much Admin controls**
   The LeadStake contract has too many admin rights which can be misused intentionally or unintentionally, i.e. in the case where the admin's private key gets compromised. Admin has the right to change all crucial parameters of smart contract like **stakingTaxRate**, **registrationTax**, **weeklyROI**, **unstakingTaxRate**, **minimumStakeValue** and **referralTaxAllocation**.
   In a scenario, these unstaking taxes can be set too high so that the user won't be able to claim back their Lead token stakes.

   Recommendation:
   Consider adding some hardcoded minimum and maximum values for these variables so that the admin cannot set values beyond the hardcoded range.

4. **Users cannot stake again within 7 days**
   Due to the time restrictions in the LeadStake smart contract's **stake**() function, users cannot stake again within a seven days period. The developer comments mentions that this condition is used to introduce a delay between user's last payout and stake but it also prevents users to stake again within 7 days

   Recommendation:
   Consider removing the time condition from the **stake**() function if this condition is unintended.

5. **No input checking in unstake() function**
   The **unstake**() function in LeadStake.sol takes amount as an input parameter from the user. The validity of this variable is not checked in the function. In a case where the user supplies zero(0) as amount, this function behaves similar to **withdrawEarnings**() function. Since **withdrawEarnings**() is a separate dedicated function for claiming all earnings, the mentioned scenario is entirely unintentional.

   Recommendation:
   Consider adding a check for input amount.

6. **In adminWithdraw() balance check is incorrect**
   In **adminWithdraw**() at line 306, the contract checks the Lead token balance of msg.sender/admin, which is inappropriate. In a case where the admin's Lead token balance is less than the amount of Lead tokens he wants to pull out from the system, the function call will revert.

   Recommendation:
   Consider removing the require statement as the Safemath already reverts when someone transfers funds more than he possesses. Or replace **msg.sender** with **address(this)**.
   Also have a look at Issue - 2 in High severity issues.

7. **SupplyPool() implementation is incorrect.**
   The **supplyPool**() function of LeadStake contract contains a statement in **For** loop (line 306) which is not implemented correctly, Even after iterating over the entire **stakeHolders[]** array, the **totalClaimable** will only store the last value for **i**(index) in the array which is incorrect.
   Also unnecessary balance checks are implemented in line 325, 327. These conditions are already handled by Safemath.
   Also Lead token can also be supplied to LeadStake smart contract directly via Lead token's **transfer**() function. So there won't be any need for **supplyPool**() function.

   Recommendation:
   Consider reimplementing the statement to account for all the addresses in **stakeHolders[]** array.

8. **ERC20 functions should be wrapped in require**
   The return value of standard ERC20 functions is neglected in LeadStake contract at line 308, 329 and 254. As per standards, erc20 function calls should be wrapped inside a **require** statement to handle token transfer failures.

   Recommendation:
   Consider wrapping all erc20 calls in require.

Low severity issues

1. **Spamming by staking minimum value of Lead Token**
   In a hypothetical (but possible) scenario users can spam the LeadStake contract by just staking the **minimumStakeValue** and yielding the referral bonus. This can be a case when the referral reward is sufficiently high.
   Let's assume the **minimumStakeValue** is 100 Lead tokens, instead of staking 1000 Lead tokens directly, the user can stake 100 Lead tokens from 10 different accounts which on registration can generate referral rewards for one/many addresses.
   If the referral rewards are high enough to cover for all staking/registration taxes, it can elevate the DoS issue mentioned above (HIgh severity issue - 1).

2. **CanApprove modifier**
   The Lead token (ERC20) contract has a **canApprove** modifier which contains an unnecessary check **balances[msg.sender] >= value.** Most ethereum projects take user's approval before collecting funds from the user's wallet. This is done by calling the erc20's approve() function with 2**256 - 1 as amount. This also saves users from signing two transactions while interacting with protocol (like staking Lead token in this case).

   Recommendation:
   Consider removing the check from the modifier.

3. **No address check for to variable in ERC20's transferFrom**
   In ERC20 (Lead Token) **transferFrom**() function there is no check for ***to*** address, this can lead to some unintentional behaviour.

   Recommendation:
   Consider using OpenZeppelin's ERC20 implementation as it is already audited as well as widely used.

4. **Unnecessary balance checks in LeadStake and ERC20**
   There are multiple instances in LeadStake and ERC20 contracts where Lead token balance of user/contract is checked (for eg: line 101) which are not required. As SafeMath already takes care of integer underflows/overflows the checks are redundant and cost gas in every transaction.

   Recommendation:
   Consider removing these checks from entire contracts and use OpenZeppelin's ERC20 implementation which is already audited as well as widely used.

5. **No decimals for LEAD Token**
   There is no **decimals**() function/field for LEAD token.

6. **Events are not as standards**
   In LeadStake contract event names are not as standards. **Staked** event should be emitted instead of **OnStake**.
   Also there is no use of indexed variables in the events. Indexed variables help in filtering of events based upon parameters. This largely reduces contract monitoring on off-chain elements like frontend and scripts.

   Recommendation:
   Consider making the event's variables indexed.

7. **No separate functions to increase/decrease ERC20 allowances.**
   Follow the standard ERC20 pattern to increase/decrease allowance amount by using separate functions to overcome the frontrun attack. More details on ERC20 frontrun attack can be found at https://github.com/ethereum/EIPs/issues/738.

8. **No mechanism to deal with unforeseen bugs**
   There is no logic implemented in Lead smart contracts to deal with future unforeseen bugs/upgrades.

   Recommendation:
   Consider adding a mechanism to pause the functioning of smart contracts if any vulnerability is detected in future. Ability to upgrade the code of smart contracts may also be considered.

9. **Unused variable - totalDistributed**
   In the LeadStake contract, **totalDistributed** variable is declared but never used.

   Recommendation:
   Consider removing the variable if not required.

10. **Contract Size**
    The size of LeadStake contract is getting close to the 24KB limit imposed by ethereum blockchain. Any contract with size over 24KB cannot be deployed to ethereum mainnet.

    Recommendation:
    Consider reducing the size of the contract by removing some unnecessary variables/functions and other size optimization techniques.

Unit Test

All unit tests provided by LeadWallet are passing without any issues.

```
Contract: LeadStake
  ✓ Should transfer a balance of 10 tokens to smart contract properly
  ✓ Should NOT create a stake without registration (39ms)
  ✓ Should register a stakeholder properly (312ms)
  ✓ Should NOT create registration twice (105ms)
  ✓ Should NOT create a stake if amount is below the minimum staking value (39ms)
  ✓ Should NOT create a stake if amount is hgher than stakeholder LEAD balance (41ms)
  ✓ Should calculate earnings properly
  ✓ Should create a stake properly (159ms)
  ✓ Should NOT stake below one week of previous stake
  ✓ Should NOT ustake if not registered
  ✓ Should NOT unstake below one week
  ✓ Should NOT unstake above stake balance
  ✓ Should unstake properly (205ms)
  ✓ Should deregister stakeholder who unstakes total stakes (204ms)
  ✓ Should NOT withdraw for non-registered users (39ms)
  ✓ Should withdraw properly for registered users (74ms)
  ✓ Should NOT withdraw if below one week interval
  ✓ Should set staking tax properly
  ✓ Should set unstaking tax properly
  ✓ Should set weekly ROI properly (55ms)
  ✓ Should set registration tax properly
  ✓ Should set referral tax allocation properly
  ✓ Should set minimum stake value properly
  ✓ Should withdraw funds from owner properly (41ms)
  ✓ Should supply to the pool properly (73ms)
  ✓ Should NOT supply pool if non-owner
  ✓ Should NOT supply if pool has enough tokens (44ms)


27 passing (2s)
```

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

Coverage Report

Test coverage of LeadWallet's smart contract is not 100%.

```
----------------|----------|----------|----------|----------|----------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------|----------|----------|----------|----------|----------------|
 contracts/     |    94.44 |     62.9 |     87.1 |    94.62 |                |
  ERC20.sol     |    83.33 |       40 |    69.23 |    83.78 |... 20,21,85,93 |
  LeadStake.sol |    98.89 |    73.81 |      100 |    98.92 |            228 |
----------------|----------|----------|----------|----------|----------------|
All files       |    94.44 |     62.9 |     87.1 |    94.62 |                |
----------------|----------|----------|----------|----------|----------------|
```

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

Slither Tool Result

```
❌ LeadStake.unstake(uint256) (LeadStake.sol:180-212) ignores return value by
IERC20(lead).transfer(msg.sender,unstakePlusAllEarnings) (LeadStake.sol#192)

❌ LeadStake.withdrawEarnings() (LeadStake.sol:246-265) ignores return value by
IERC20(lead).transfer(msg.sender,totalReward) (LeadStake.sol#254)
```

Recommendations

1. In LeadStake contract the statement in **registerAndStake**()
   referralBonus = (registrationTax.mul(referralTaxAllocation)).div(100)
   Can be put inside the if block after line 113. This will save gas costs when no referrer is supplied.

2. LeadStake.sol - The statement
   require(_amount >= registrationTax.add(minimumStakeValue));
   Should be placed before the token transfer. This will save gas when the amount is less than the total taxes.

3. Unnecessary mappings
   In LeadStake contract various mappings are used to store different user data like **stakes**, **stakeRewards**, **lastClock**, etc. Instead a single mapping from an address to a Struct could be used. This will also reduce the contract size and save deployment cost.

4. ERC20 (LEAD Token) functions should be declared **external** instead of **public**. This will save transaction costs.

Concluding Remarks

While conducting the audits of LeadWallet's smart contract, it was observed that the contracts contain High, Medium, and Low severity issues, along with several areas of recommendations.

Our auditors suggest that High, Medium, Low severity issues should be resolved by the LeadWallets's developers. Resolving the areas of recommendations are 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's platform or it's products neither this audit is investment advice.

Notes:

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

ImmuneBytes Pvt Ltd.