Land LayBy Technologies Ltd.

Smart Contract Security Audit

August 2018

BUILDING TRUST
THROUGH RESULTS

HAWK
CYBER SECURITY

+1 (646) 781 8174
contact@hawksec.io
www.hawksec.io

We have completed our engagement to assess the security of your smart contracts through the performance of a smart contract security audit. The findings and recommendations resulting from our assessment are provided in the detailed findings and recommendations sections.

The procedures summarized in this report do not constitute any form of review or assurance in accordance with any generally accepted, review or other assurance standards and accordingly we do not express any form of assurance. This assessment relates to procedures that were performed at a specific point in time. As a result, it does not reflect events or circumstances that may arise after its publication.

The ratings in the detailed findings and recommendations section of this report do not represent a conclusion on the adequacy or effectiveness of internal controls. Rating definitions are as defined in Appendix A.

Sincerely,

Hawk Cyber Security

Table of Contents

Executive summary

Introduction

Land LayBy Technologies Limited ("Land LayBy") engaged Hawk Cyber Security ("HawkSec") to perform a Smart Contract Security Audit on its ERC20 Token and Token Sale smart contracts in order to discover the risk of exposure to security threats and vulnerabilities.

The HawkSec testing team ("We") performed the audit during the month of August 2018 and our procedures were limited to those described in this report.

The findings in this report result from our attempts to discover, validate and exploit vulnerabilities that were considered to be within the project's scope and duration. The recommendations provided in this report are structured to facilitate remediation of the identified security risks.

Limitations

Security assessments cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, the assessment allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract, while some lack protection only in certain areas. We therefore carry out a source code review to determine all locations that need to be fixed. Hawk Cyber Security has performed extensive auditing in order to discover as many vulnerabilities as possible.

Scope and overall summary

Smart Contract Security Audit: The smart contract security audit was a review on Land LayBy's token and token sale smart contracts (HRBE or Harambee token) based on unit testing coverage, automated vulnerability scanning and manual line-by-line inspection of the code with the goal of detecting existing vulnerabilities, taking into consideration known attacks, best coding practices and business logic issues, including conceptual consideration in relation to the published white paper and the term of the token sale documents.

The audited code was provided by Land LayBy and is located in the public repository https://github.com/LandLayby/SmartContract and the commit version is `1a5157e142a93d2e15e9203cc9f1207768154945`.

The following files were included in the scope of this audit:

File	Fingerprint (SHA-256)
Math\Math.sol	0645e4072edd04dc75bbbe37812ffc5f53c4e778f58f3d61051745051d013941
Math\SafeMath.sol	2bbbd91f5fc8ad1ece8c48adce20854f434c9dacac75dc38b62cba2e5ab9fb5b
Token\ERC20.sol	b56deb8ca835d8eec4cdb6aba0926b0c341f06817be8ba3632482e09a4f0246c
Token\HarambeeToken.sol	52fd3387f739e27825870c7b85df861761a2f877690df97b8c9682a13dac2641
Token\Owned.sol	be942ed48e047b02bac606c41feb1b082396b44cd3bb5adb114d7855ac55d783
Token\Pausable.sol	dbe6e23226fe746a0c2754ebdb4060f02e0e71c9fa763990e69bddb79df1a80e
TokenSale\HarambeeTokenSale.sol	74c6a644a5fb4161cc1c26ea695eeb58b30a19a8da8b5398c8267539b4ba6c7a
Vesting\TokenVesting.sol	bb5cf5a3676e774fe4b09fbdecc6aa173eb7428477b273913144a8cef39ba79c

Overview of Land LayBy's Project

Land LayBy is a unique real estate firm company with an active financial technology arm that specializes in solving land acquisition problems in developing countries. The Land LayBy Listing (LLL) platform is a blockchain-based decentralized and secure ledger for registering land and conveyance data that can never be altered, corrupted, forged or replicated in error.

The platform is powered by the Harambee (HRBE) token.

Harambee Token Smart Contract

Token details

Description	Value
Decimals	18
Smallest Unit	10^{-18}
Maximum token supply	1,000,000,000
Percentage of tokens for sale	Pre-sale: 20% (200M) Main Sale: 30% (300M)
Minimum token purchase	20 HRBE
Maximum token purchase	50,000,000 HRBE
Token Distribution	Post ICO
KYC	Yes (whitelist, through platform)
Burnable	Yes
Lock Mechanism	Yes (manually lifted)
Vesting	Yes
Mintable	No (Fixed total supply)

Summary of Findings

Overall, the code is clearly written, and demonstrates effective use of abstraction, separation of concerns, and modularity. The contracts were originally copied from the OpenZeppelin repository and the code was modified slightly to adapt to the project's needs. Both Token and Token Sale contracts were strategically kept to a minimum amount of functionality in order to reduce potential security issues by minimizing the attack surface.

Land LayBy's development team demonstrated high technical capabilities, both in the design of the architecture and in the implementation.

No critical nor high issues were identified while five low technical issues, mostly related to code styling and best practices, required the attention of Land LayBy's team. In addition, two conceptual issues, related to the statements expressed on the whitepaper were identified.

All issues were promptly fixed or addressed by Land LayBy's team.

We rated the findings based upon the risk they pose to the company. Please refer to Appendix A – Risk rating definitions for our rating explanation.

Unit Testing Coverage

The testing team made a review of the unit testing that was performed by the development team, in order to evaluate the maturity of test cases of the contracts.

In addition, the testing team reproduced each and every test case to validate the results and match them against the original test.

In conclusion, the smart contracts have shown a great test coverage and unit testing, having covered 95% of the functions.

The following are the tested cases:

Harambee Token contract

✓ The checkmark indicates that the test case returned the expected results.

Function	Test Case
Contract Deployment	✓ Tested deployment of the Harambee Token on an ETH wallet and performed the basic ERC 20 functions.
Transfer	Transfer token to various addresses and check debit and credit operation for the accounts: ✓ Transfer tokens to invalid addresses and verify the error messages. ✓ Test Debited account and credit account of token sender and receiver

Function	Test Case
	✓ Test if the amount from the Sender's account is debited at first and then credited in the receiver's account
Approve	✓ Test Approve functionality where Sender (Account 1) allows Account2 to transfer specific number of tokens to the Account3, from the Sender's Account (Acc1). ✓ Test if the owner account (Account1) and the account permitted (Account2) for transactions are not the same account
Transfer From	✓ Test if the account permitted to do transactions, can transfer token to any other account (as the Owner account has permitted it to do so) ✓ Test if the Transaction is completed ✓ Test if the permitted account can transfer only the specified number of tokens as granted to it and not exceeds the amount permitted. ✓ Test if the permitted Account can transfer the lesser amount than the specified one. ✓ Test if the owner Account and permitted Account should not be same which can Transfer Money to any Receivers account
Burn	Burn the tokens supply for reducing the size of tokens and perform boundary value analysis testing on the same: ✓ Test if the Burning of Tokens is Executed only by owner Account ✓ Test above functionality after transferring the ownership and check with new owner too ✓ Test if other available accounts cannot burn the existing Tokens
Transfer Ownership	Transfer the ownership from main owner to new owner and check all the rights of ownership for new owner: ✓ Test if the Previous owner still cannot perform the functions, the one which only Owner can perform ✓ Test if the New owner can transfer its ownership to any other account or the Previous Owner too
Ether Transfer	Transmission of ETH should be allowed in between 2 accounts: ✓ Test if transfer of Ether consumes few percent of Gas on every transaction ✓ Test if transfer of ETH is allowed with Zero Gas too

Harambee Token Sale contract

✓ The checkmark indicates that the test case returned the expected results.

Function	Test Case
Deployment of the Token Sale Contract	✓ Deployment of the token sale contract with the Harambee Token and funds collection addresses as input parameters. ✓ Test ownership of Admin functions of the contract.
UNIX Time	✓ Test the UNIX Time parameter on deployment time. ✓ Test Pre-sale start and end time ✓ Test Main Sale start and end time. ✓ Test if start and end time of both pre-sale and main sales is mentioned correctly once the Contract is Deployed.
Transaction Allowed Time for Pre-Sale	✓ Test transfer ETH from various accounts to token sale contract address before the sale starts. ✓ Test error messages thrown by ETH wallet and check if ETH can be sent to contract or not
Transactions	✓ Test sending 2 ETH from Account1 to Token Sale Contract and check balance of Account1, which is debited by 2 ETH and also check the fund collector wallet whether the 2 ETH are credited.
Contributor's Address	✓ Test the validity of the address of the contributor (Not NULL) and send tokens from invalid address.
Token Transaction	✓ Test Transfer Token from main Token Contract account to Token Sale Contract and check the total supply of tokens.
Wei	✓ Test "Wei Raised" value after each and every contributor transfer ETH.
Limitation of Token Supply	✓ Test transferring more tokens than the total supply from sales contract to contribution address.
Pre-sale Execution	✓ Test Pre-sale activation/deactivation, executed as owner.
Contribution Allowed	✓ Test if IsContributionAllowed validation performed properly for all cases, including when pre-sale and main sale times ended.
ETH Transaction	✓ Test is ETH contributions are allowed before start-site of pre-sale and main sale.
Transfer Ownership	✓ Test if previous owner can perform owner-only functions. ✓ Check if the New owner can transfer its ownership to

Function	Test Case
	any other account or the Previous Owner too
Token Transfer to Contributors	✓ Test if owner is able to transfer the tokens to contributors' accounts while sale is in progress and after the end of pre-sale and main-sale period.

Know Attacks and security issues on solidity smart contracts

The following is a summary of the checked known attacks and issues and the results in the tested contracts:

Reentrancy: Not vulnerable

Any interaction from a contract A with another contract B and any transfer of Ether hands over control to contract B. This makes it possible for B to call back functions belonging to contract A before this interaction is completed. Furthermore, it's important to take multi-contract situations into account. The called contract (B) could modify the state of a third contract (C) it depends on.

https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

Integer Overflows & Underflows: Not vulnerable

An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of the type. For example, if you store your number in the uint256 type, it means your number will be stored in a 256 bits unsigned number ranging from 0 to 2^256. The most common result of an overflow is that the least significant representable bits of the result are stored; the result is said to wrap around the maximum (i.e. modulo power of two).

An overflow condition gives incorrect results and, particularly if the possibility has not been anticipated, can compromise a program's reliability and security.

https://ethereumdev.io/safemath-protect-overflows/

Timestamp dependence: Not Found

Every block has its own parameters such as timestamp. Timestamp can be manipulated by miners, and that way cause problems with synchronization. Timestamp should not be used for main components of the system.

https://github.com/ethereum/wiki/wiki/Safety#timestamp-dependence

Gas limits and loops: Found (Low - Finding L2)

Gas limit and loops are dangerous for a project. Gas limit detects amount of resources for completion of smart contract code. If a smart contract contains loops, there is a change the contract with stall, depending on the assigned amount of gas.

This means the system will stop working because the limit of resources (gas) is reached.

http://solidity.readthedocs.io/en/develop/security-considerations.html#gas-limit-and-loops

DoS with unexpected throw: Not vulnerable

Dos attacks are very popular nowadays. They are meant to stop smart contract functioning. This can happen if the smart contract's source code has vulnerabilities and they are not detected and repaired.

https://github.com/ethereum/wiki/wiki/Safety#dos-with-unexpected-throw

Dos with block gas limit: Not vulnerable

Blocks have not only timestamp parameters, but many more. One of them is the amount of computations performed. If the limit of computational operations has been exceeded, then the iteration will fail. Every failed iteration will cost gas anyway, that's especially bad if an attacker has access to the gas price of smart contract. This can lead to stealing resources or exceeding gas limit.

https://github.com/ethereum/wiki/wiki/Safety#dos-with-block-gas-limit

Transaction-ordering dependence: Not vulnerable

While iterations are made they are being held in a queue. This means the more iterations are made, the longer the wait-time. That is not acceptable for systems with large amount of users.https://github.com/ethereum/wiki/wiki/Safety#transaction-ordering-dependence-tod

TX.Origin: Not found

This is not the recommended method for implanting authorization and it shouldn't be used.

http://solidity.readthedocs.io/en/develop/security-considerations.html#tx-origin

Exception disorder: Not vulnerable

There are situations when an exception could be raised, for example, if the execution runs out of gas, if the call stack reaches the limit or when the 'throw' command is executed. Security of contracts can possibly be affected by irregularities of how exceptions are being handled.

https://eprint.iacr.org/2016/1007.pdf

Gasless send: Not found

If the gas limit is reached and an iteration is sent for completion without payment, a problem will occur. The main reason for this issue is when developers do not prepare for transferring "ether" in the code. The problem occurs when the function C.send(amount) is being compiled with an empty signature.

https://eprint.iacr.org/2016/1007.pdf

Balance equality: Not found

An assert guard triggers when an assertion fails - such as an invariant property changing. For example, the token to ether issuance ratio, in a token issuance contract, may be fixed. This can be verified at all times with an assert(). Assert guards should often be combined with other techniques, such as pausing the contract and allowing upgrades. (Otherwise, you may end up stuck, with an assertion that is always failing.)

https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/recommendations.md#enforce-invariants-with-assert

Transfer forwards all gas: Not found

When sending ether, awareness of the relative tradeoffs between the use of `someAddress.send()`, `someAddress.transfer()`, and `someAddress.call.value()()` is needed.

Using `send()` or `transfer()` will prevent reentrancy but it does so at the cost of being incompatible with any contract whose fallback function requires more than 2,300 gas.

https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/recommendations.md#be-aware-of-the-tradeoffs-between-send-transfer-and-callvalue

The smart contracts were also validated against the following common issues:

- ERC20 API violation - Not Vulnerable
- Malicious libraries - Not Vulnerable
- Compiler version not fixed - Not Vulnerable
- Redundant fallback function - Not Vulnerable
- Send instead of transfer - Not Vulnerable
- Unchecked external call - Not Vulnerable
- Unchecked math - Not Vulnerable
- Unsafe type inference - Not Vulnerable
- Implicit visibility level - Not Vulnerable

- Address hardcoded - Not Vulnerable
- Using delete for arrays - Not Vulnerable
- Integer overflow/underflow - Not Vulnerable
- Locked money - Not Vulnerable
- Private modifier - Not Vulnerable
- Revert/require functions - Not Vulnerable
- Using var - Not Vulnerable
- Visibility - Not Vulnerable
- Using blockhash - Not Vulnerable
- Using SHA3 - Not Vulnerable
- Using suicide - Not Vulnerable
- Using throw - Not Vulnerable
- Using inline assembly - Not Vulnerable

Detailed findings and recommendations

The following section details the assessment findings and provides recommendations that Land Layby should consider implementing in order to remediate the identified security issues.

Finding M1: Incorrect tokenSupply value

Impact: Medium

Status: FIXED

This issue has been fixed.

Description

The Harambee Token contract declares a tokenSupply of 500 million tokens. This is different from the value stated in the published whitepaper: 1 Billion.

Proof of concept

HarambeeToken.sol

```
8   //This is the Main Harambee Token Contract derived from the other two contracts Owned and ERC20
9   contract HarambeeToken is Owned, ERC20 {
10
11      using SafeMath for uint256;
12
13      uint256  public tokenSupply = 500000000;
14
15      //This notifies clients about the number of tokens minted
16      event TokensMinted(address owner,uint256 value);
```

Instances

HarambeeToken.sol:13

Recommendation

The contract should be updated in order to set 1 Billion as tokenSupply.

Finding M1: Token contract makes use of Mint function

Impact: Medium

Status: FIXED

This issue has been fixed.

Description

The Harambee Token contract contains the `mintTokens` owner-only function that allows the owner to mint new tokens manually. This function is not really necessary, since the whitepaper states that the token will be created with a total token supply of 1 billion, and no more tokens should be minted.

By including the `mintTokens` function the social contract with the contributors is affected.

Proof of concept

HarambeeToken.sol

```
30      /* This function is used to mint additional tokens
31       * only admin can invoke this function
32       * @param _mintedAmount amount of tokens to be minted
33       */
34      function mintTokens(uint256 _mintedAmount) public onlyOwner {
35          balanceOf[owner] = balanceOf[owner].add(_mintedAmount);
36          totalSupply = totalSupply.add(_mintedAmount);
37          emit TokensMinted(owner,_mintedAmount);
38      }
```

Instances

HarambeeToken.sol:13

Recommendation

Remove the Mint function from the contract.

Finding L1: Defining constructors as same-name functions is deprecated

Impact: Low

Status: FIXED

This issue has been fixed.

Description

The Harambee Token Sale contract is using compiler version 0.4.24, which throws a warning compilation error. The contract makes use of an outdated constructor function declaration format.

Proof of concept

HarambeeTokenSale.sol

```
50  function HarambeeTokenSale(HarambeeToken _addressOfRewardToken, address _wallet) public {
51    require(presalestartTime >= now);
52    require(_wallet != address(0));
53
54    token = HarambeeToken (_addressOfRewardToken);
55    wallet = _wallet;
56
57    owner = msg.sender;
58  }
```

Instances

HarambeeTokenSale.sol:50

Recommendation

The contract should be updated in order to follow the chosen version's coding standards and style.

Finding L2: Gas Optimization (Costly Loop)

Impact: Low

Status: Risk Accepted

The team decided to accept the risk of this issue due to the fact that the function is owner-only, and it does not affect the proper functioning of the token sale.

Description

The Harambee Token Sale contract includes the `manualBatchTransferTokens` function, which contains a `for` loop statement that receives an array as parameter.

If there are many items in the given array, the execution of the function will fail, and an out-of-gas exception will be thrown.

Proof of concept

HarambeeTokenSale.sol

```
125
126   //This function is used to do bulk transfer token to contributor after successful audit manually
127     function manualBatchTransferToken(uint256[] amount, address[] wallets) public onlyOwner {
128         for (uint256 i = 0; i < wallets.length; i++) {
129           token.transfer(wallets[i], amount[i]);
130           emit TokensTransferred(wallets[i], amount[i]);
131         }
132     }
133
134   }
```

Instances

HarambeeTokenSale.sol:127

Recommendation

Make use of the function with care in order to avoid exceptions.

Finding L3: Improper validation on fallback function

Impact: Low

Status: FIXED

This issue has been fixed.

Description

The Harambee Token Sale contract's fallback function performs a validation on the contribution amount. This validation is not properly made and can enable potential integer overflow scenarios.

Proof of concept

HarambeeTokenSale.sol

```
60  // fallback function  used to buy tokens , this function is called when anyone sends ether to this contract
61  function ()  payable public {
62
63    require(msg.sender != address(0));                    //contributors address should not be zero
64    require(msg.value != 0);                              //contribution amount should be greater then zero
65    require(isContributionAllowed());                     //Valid time of contribution and cap has not been reached
66
67    //forward fund received to Harambee multisig Account
68    forwardFunds();
```

Instances

HarambeeTokenSale.sol:64

Recommendation

Use the great than or equal (>=) operator on the affected line.

Finding L4: Invoking events without emit prefix is deprecated

Impact: Low

Status: FIXED

This issue has been fixed.

Description

The Harambee Token contract is using compiler version 0.4.24, which throws a warning compilation error. The contract does not use the 'emit' prefix before the event function call.

Proof of concept

HarambeeTokenSale.sol

```
73
74      //Notify server that an contribution has been received
75      ContributionReceived(msg.sender,msg.value);
76  }
77
```

Instances

HarambeeTokenSale.sol:75

Recommendation

The contract should be updated in order to follow the chosen version's coding standards and style.

Finding L5: Approve function lacks proper validation

Impact: Low

Status: Risk Accepted

The team decided to accept the risk of this issue due to its unlikelihood.

Description

The 'approve' function in ERC20 contracts enables a token holder to "set aside" certain number of tokens from his balance to be used by another token holder through the use of the 'transferFrom' function.

By using the implemented method, there is a small period of time in which a malicious recipient is able to receive funds from the sender multiple times.

This attack can only be performed against individual users and does not affect the token ecosystem directly.

Proof of concept

```
77      /* This function allows _spender to withdraw from your account, multiple times, up to the _value amount.
78      * If this function is called again it overwrites the current allowance with _value.
79      * @param _spender address of the spender
80      * @param _amount amount allowed to be withdrawal
81      */
82      function approve(address _spender, uint256 _amount) public returns (bool success) {
83          allowed[msg.sender][_spender] = _amount;
84          emit Approval(msg.sender, _spender, _amount);
85          return true;
86      }
87
88      /* This function returns the amount of tokens approved by the owner that can be
89      * transferred to the spender's account
90      * @param _owner address of the owner
91      * @param _spender address of the spender
92      */
93      function allowance(address _owner, address _spender) public view returns (uint256 remaining) {
94          return allowed[_owner][_spender];
95      }
96  }
```

Instances

Function 'approve' in ERC20.sol

Recommendation

It is recommended to validate if the 'allowed' value of a spender is 0, and stop the execution if not.

Conclusions

The smart contract has been analyzed under both technical and conceptual aspects, with different open-source and proprietary tools and a line-by-line manual inspection has been made. Overall, we found that the Harambee Token and Token sale contracts employ very good coding practices and has clean, documented code.

We have no remaining security concerns about the Harambee Token and Token Sale smart contracts, as all detected issues were either fixed or addressed.

Appendix A – Risk rating definitions

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Risk Level	Value	Required Action
Critical	9 - 10	Immediate action to reduce risk level.
High	7 - 8.9	Implementation of corrective actions as soon as possible.
Medium	4 - 6.9	Implementation of corrective actions in a certain period.
Low	2 - 3.9	Implementation of certain corrective actions or accepting the risk.
Informational	0 - 1.9	An observation that does not determine a level of risk.

Common Vulnerability Scoring System (CVSS) is a free and open industry standard for assessing the severity of computer system security vulnerabilities. It is under the custodianship of NIST. It attempts to establish a measure of how much concern a vulnerability warrants, compared to other vulnerabilities, so efforts can be prioritized. The scores are based on a series of measurements (called metrics) based on expert assessment.

The detailed methodology of the computation process for CVSS version 3.0 is available here: https://www.first.org/cvss/specification-document.