HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Lingo
Date: 08 Nov, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Lingo
Approved By	Niccolò Pozzolini | Lead Solidity SC Auditor Maksym Fedorenko | Co-Auditor Paul Fomichov | Approver
Tags	ERC20 token; Staking
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://www.mylingo.io/
Changelog	08.11.2023 - Initial Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by Lingo (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Lingo is a staking protocol composed by the following contracts:

- *Lingo* — ERC-20 token that mints all initial supply to a deployer. The token has fee on transfer functionality, which allows to charge up to 5% fee on every transfer if sender or recipient is not whitelisted. Additional minting is allowed. Burning is allowed.
  It has the following attributes:
  - Name: Lingo
  - Symbol: LINGO
  - Decimals: 18
  - Initial supply: 1 Billion.
- *Distributor* — a contract that rewards users for staking their tokens. The admin defines the time frames during which users might stake the tokens, when the timeframe ends the admin deposits tokens, which might be claimed proportionally to the users distribution during the finished staking period.

Privileged roles

- The admin of the *Lingo token* contract can arbitrarily add, delete and modify the addresses stored. It is, therefore, entitled to impersonate or change the logic of critical components of the system at will.
- The admin of the Distributor contract can:
  - set the staking timeframe;
  - withdraw fees up to 5%;
  - distribute arbitrary amount of tokens to the users;
  - take unclaimed tokens after some period of time.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are detailed.
- Technical description is robust.

Code quality

The total Code Quality score is **3** out of **10**.

- Presence of code duplications.
- Gas inefficient for loops
- Use of hard coded numbers instead of constants.
- Code duplications

Test coverage

Code coverage of the project is **82.2%** (branch coverage).

- Some multistep interactions are not tested.
- Lacking test for non whitelisted *Distributor* contract.

Security score

As a result of the audit, the code contains **3** critical, **1** high, **3** medium, and **4** low severity issues. The security score is **0** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **1.6**. The system users should acknowledge all the risks summed up in the risks section of the report.

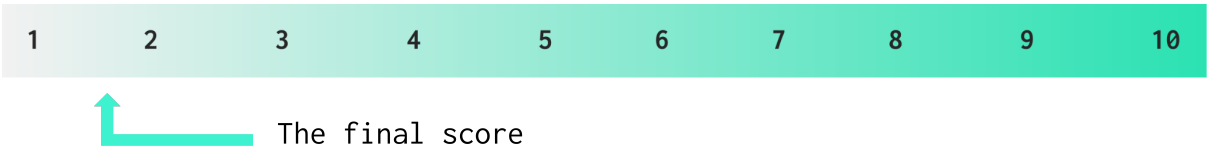

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
8 November 2023	4	3	1	3

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Risks

- The LINGO token is a fee on transfer tokens; this means that if the user or contract is not whitelisted, fees are applied on each transfer.
- If the decentralized exchange protocols contracts are not whitelisted, the Lingo token would not be able to be traded if they do not support “fees on transfer” tokens.
- The whitelisting and de-whitelisting is controlled by the admin of the token contract.
- According to the documentation, the distribution slot timeframe is one month, but it might be updated to an arbitrary amount of time.
- The total reward for the deposited tokens is defined by the amount of tokens, which admin distributes to the contract and is not granted to be greater than zero.
- The withdrawal fees from the Distributor contract might be updated any time up to the 5%, and might be taken in addition to fees on transfer with the maximum value of 5%.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Findings

■■■■ Critical

C01. Total Supply Violated

Impact	High
Likelihood	High

The documentation states that the total supply is 1 billion tokens, however it is not the “total supply”, but the “initial supply”, as the amount of tokens might be changed by minting.

The situation is acknowledged by the developers, since the *mint()* function is properly documented in the technical requirements, but this wrong wording might lead to users' misunderstandings.

```
/**
  * @dev Mint new tokens.
  * @param to The address to mint the tokens to.
  * @param amount The amount of tokens to mint.
  */
 function mint(address to, uint256 amount) external onlyOwner {
   _mint(to, amount);
 }
```

Path: ./contracts/Lingo.sol

Recommendation: Update the documentation to remove the confusion between the initial and total supply.

Found in: bbd2749

Status: New

C02. Funds Lock After Admin Claims The Rewards

Impact	High
Likelihood	Medium

The system allows the admin to claim unclaimed tokens after some period of time with *adminClaim()* function. The function sets the *remainingTokensToClaim* for specific period to 0, but the *totalCredits* and *monthlyProfit* values and *user.balance* stays the same.

During the future users claim the claimReward function calculates the *credits* to distribute with *claim* amount based on *totalCredits* and *monthlyProfit*.

*claim = (credits * _distributionHistory[i].monthlyProfit) / _distributionHistory[i].totalCredits;*

This non zero value is being subtracted from the *remainingTokensToClaim* value, which leads to the arithmetic overflow because the *remainingTokensToClaim* was set to *0*:

_distributionHistory[i].remainingTokensToClaim -= claim

Due to the *senderDetails.lastClaimedTimestamp* which is smaller than *lastDistributionDetails.endTime* and non zero *user.balance* the *havePendingClaim* modifier reverts and does not allow to withdraw or deposit tokens.

This leads to the user funds lock and inability to withdraw, deposit or claim reward tokens.

```
  function adminClaim() external onlyOwner nonReentrant {
    uint256 totalClaim = 0;

    /// Calculates the total amount of tokens that can be claimed by the owner.
    for (uint256 i = 0; i < _distributionHistory.length; i++) {
      if (((block.timestamp / 3600) - _distributionHistory[i].endTime) >=
_adminClaimPeriod) {
        totalClaim += _distributionHistory[i].remainingTokensToClaim;
        _distributionHistory[i].remainingTokensToClaim = 0;
        // Missing: _distributionHistory[i].totalCredits = 0;
      }
    }

   [...]
  }
```

Steps to Reproduce:

1. A user deposits funds and does not claim tokens during the slot.
2. The admin claims the unclaimed rewards.
3. The user is not able to withdraw funds from the contract even when the new distribution rounds are introduced.

Proof of Concept:

```
const amountBN = BN(100).mul(BN(10).pow(DECIMALS_BN));
await deposit(distributionContract, token, user1, amountBN);
await skipTime(SLOT_BN.mul(BN(3600)).add(BN(3600)).toNumber());
const distributeAmountBN = BN(10000).mul(BN(10).pow(DECIMALS_BN));
await distribute(distributionContract, token, distributeAmountBN);

const adminClaimPeriod = await distributionContract.getAdminClaimPeriod();
```

```
await skipTime(adminClaimPeriod.mul(BN(3600)).toNumber());

await distributionContract.adminClaim();

// @dev we should fill the gap between the current `block.timestamp` and
`_currentSlotStart`.
// the amount of slots to add depends on the `_slot` value and admin claim
period
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);
await distribute(distributionContract, token, distributeAmountBN);

await expect(distributionContract.connect(user1).claimRewards()).to.be.reverted;

const userStats = await distributionContract.getUserStatus(user1.address);
await
expect(distributionContract.connect(user1).withdraw(userStats.balance)).to.be.re
vertedWith("LINGO: User have unclaimed tokens. Please claim it before deposit or
withdraw")
```

Path: ./contracts/Distributor.sol : claimRewards(), adminClaim(), withdraw(), deposit()

Recommendation: One of the possible solutions is to modify the *adminClaim()* function to set *totalCredits* to 0 for the claimed slots. This will prevent the *claimRewards()* function from trying to collect the rewards related to these slots, since totalCredits are checked to be > 0 before the collect operations.

Another possible solution is to rework the implementation of the *claimRewards* function to add the conditional statement within the for loop, which checks if the *remainingTokensToClaim* of the distribution round is not zero and skips the iteration otherwise.

Found in: bbd2749

Status: New

C03. Broken Fees Management Leads To Excessive Deposit Accounting

Impact	High

Likelihood	High

The Distribution contract does not handle fees on transfer tokens functionality properly when collecting the user deposits.

Instead of passively computing the transferred amount by comparing the contract balance before and after the *transferFrom()* operation, the Distribution contract manually subtracts the fees to be paid, creating multiple issues:

- The contract does not check whether the sender account is included in the ExternalWhitelist, leading to a smaller deposit than the intended one.
- If the Distribution contract is not whitelisted, the fees get subtracted from the input variable amount at line 260. This reduced amount is then used in the allowance check at line 268; Since the fees are collected by the Lingo token contract, this check is wrong because the allowance amount should also include the fees; otherwise, the transfer will revert.
- If the Distribution contract is not whitelisted, the fees get subtracted from the input variable amount at line 260. This reduced amount is then used in the *transferFrom()* at line 304; Since the fees are collected by the Lingo token contract, this check is wrong because the transfer amount should also include the fees; otherwise, the deposited amount will be smaller than the intended one.
- Since the fees accountancy is happening in the wrong contract, during the deposit if the Depositor contract and sender user are not whitelisted, the amount of deposited tokens is recorded before the transfer fees are applied by the Lingo token contract.
  As a consequence, the accounted funds on the Distribution contract will be higher than the actual deposited tokens, leading to:
  - Wrong rewards allocation
  - Inability for the last users to withdraw to receive the full owed amount - a portion or the whole amount, depending on the protocol usage and the deposited amounts. In such cases, users would need to check the Lingo balance of the Distribution contract to get the maximum withdrawable amount, since the contract state would be inconsistent.
    The withdrawal process would, in practice, work like a Ponzi scheme.

Steps to Reproduce:

1. User and Distributor contracts should not be whitelisted.
2. User deposits tokens to the Depositor Contract
3. User checks his balance on the contract with the *getUserStatus()* function and attempts to withdraw this amount.

This leads to the transaction revert with a *ERC20: transfer amount exceeds balance* error.

Proof of Concept:

```
const user1amountBN1 = BN(100).mul(BN(10).pow(DECIMALS_BN));
const user1expectedAmountAfterFee1 = await debitFee(token, user1amountBN1);
await token.connect(user1).approve(distributionContract.address,
user1amountBN1);

await expect(distributionContract.connect(user1)
.deposit(user1amountBN1)).to.emit(distributionContract, 'Deposit')
.withArgs(user1.address, user1expectedAmountAfterFee1);
const userStats = await distributionContract.getUserStatus(user1.address);

skipTime(BN(15 * 24).mul(BN(3600)).toNumber());

await expect(distributionContract.connect(user1)
.withdraw(userStats.balance)).to.be.revertedWith("ERC20: transfer amount exceeds
balance");
```

Path: ./contracts/Distributor.sol : deposit()

Recommendation: Instead of manually computing the fees to apply, the *deposit()* function should derive the actual transferred amount by comparing the Distribution contract balances before and after the *transferFrom()*. To make it possible, the *transferFrom()* should then be moved at the beginning of the function.

Found in: bbd2749

Status: New

■ ■ ■ High

H01. Inability To Withdraw Funds and Claim Rewards

Impact	High
Likelihood	Medium

The modifier *isActive* is applied to the *withdraw()* function, locking the user funds if the administrators do not behave as expected.

The *isActive* modifier makes sure that the admins have distributed the rewards for the last passed slot. Until they do it, the users will not be able to withdraw the funds.

The same applies for the *claimRewards()* function: even if the last slot has not yet been funded with rewards, the users should be able to claim their rewards related to the previous slots, which is currently not possible.

The *withdraw()* function also implements the *havePendingClaim* modifier, which is reverting the withdraw transaction if the user have pending rewards to claim. Since the *claimRewards()* function cannot always be executed, it results in another block to users' funds withdrawal.

Steps to Reproduce:

1. The user deposits funds with the *deposit()* function.
2. Wait after the last slot has ended.
3. Call the *withdraw()* function.
4. The transaction will revert with an error *LINGO: Distribution is on hold. Please contact the admin*.

Path: ./contracts/Distributor.sol : withdraw(), claimRewards()

Recommendation: Users should always be able to withdraw their funds, even if it comes at the costs of discarding the unclaimed accrued rewards. Besides fixing the *withdraw()* function, another option is to implement an emergency withdraw function to achieve this purpose.

The *claimRewards()* function must be fixed to always allow the users to claim the collected rewards related to previous slots.

Found in: bbd2749

Status: New

Medium

M01. Denial Of Service

Impact	**High**
Likelihood	**Low**

The system has multiple permanently growing storage arrays. If the arrays are populated with a significant amount of elements, operations on them could incur in the block gas limit and revert the transactions, leading to Denial of Service.

The arrays to which the issue applies are:

1. In the Distribution contract, the array *_distributionHistory* which is used in the *claim()* and *adminClaim()* functions.
2. In the Lingo contract, the arrays *_internalWhitelistedAddresses* and *_externalWhitelistedAddresses* which are used in the token de-whitelisting functionality through the function *_removeFromInternalWhiteList()* and *_removeFromExternalWhiteList()*.

Paths:

./contracts/Lingo.sol : removeFromWhiteList(), _removeFromExternalWhiteList(), _removeFromInternalWhiteList(), getExternalWhitelistedAddresses(), getInternalWhitelistedAddresses()

./contracts/Distributor.sol : claimRewards(), adminClaim(), getDistributionHistory(), getUserAddresses()

Recommendation: Different solutions can be applied in the two problematic situations:

- In the Distribution contract, a storage mapping should be implemented to store each users' last claimed slot, so that during the claim process it will not be needed to cycle the whole *_distributionHistory* array from the first element. On top of that, it is required to implement a pagination functionality on both the *claim()* and *adminClaim()* functions, so that users will be able to claim the accrued rewards spread over a large number of slots.
- In the Lingo contract, the mappings *_isInternalWhiteListed* and *_isExternalWhiteListed* instead of a bool could contain the users' indexes in the arrays *_internalWhitelistedAddresses* and *_externalWhitelistedAddresses*, so that during the whitelist removal operations it will no be required to cycle over these entire arrays.
  Note: the user indexes should be saved with a +1 offset, so that the zero value means that the users are not whitelisted.

Found in: bbd2749

Status: New

M02. Inefficient Gas Model For Deposit and Withdraw Operations

Impact	Low
Likelihood	High

The storage variable *userDetails* is not loaded into memory, and is read and written many times throughout the *deposit()* and *withdraw()* functions, costing the user much more gas than it should.

Moreover the math in *withdraw()* and *deposit()* operations includes redundant operations, which could be heavily simplified. The redundant operations are now going to be explained for the *deposit()* function, but the same applies for *withdraw()*:

1. *userDetails.forecastedCredits* gets subtracted by the remaining forecasted credits for the current slot:

   ```
   userDetails.balance * (_currentSlotEnd - block.timestamp / 3600)
   ```

2. The balance gets increased by the deposited amount: `userDetails.balance += amount`

3. With the updated user balance, *userDetails.forecastedCredits* gets increased by the remaining forecasted credits for the current slot:

```
userDetails.balance * (_currentSlotEnd - block.timestamp / 3600)
```

The operations on the *userDetails.forecastedCredits* can be replaced by a single add, considering the deposited amount instead of the whole user balance:

```
userDetails.forecastedCredits += amount * (_currentSlotEnd - block.timestamp / 3600);
```

Paths:

./contracts/Distributor.sol : deposit(), withdraw()

Recommendation:

- Load the *userDetails* storage variable into memory to be read and manipulated, and write it back to storage at the end of the functions.
- Remove the mentioned redundant math operations.

Found in: bbd2749

Status: New

M03. Admin Can Change _adminClaimPeriod And Collect All The Users' Unclaimed Rewards

Impact	Medium
Likelihood	Medium

Admins can withdraw all the uncollected funds by calling *adminClaim()* right after updating the *_adminClaimPeriod* to a very small value.

When *_adminClaimPeriod* gets updated, users need to have a heads-up time window to collect their rewards in case of a malicious update from the admins.

Steps to Reproduce:

1. The protocol has been active for a while and the contract is holding users' unclaimed rewards
2. A malicious administrator calls *updateAdminClaimPeriod()* providing an extremely low new value
3. The malicious administrator calls *adminClaim()* to collect the unclaimed rewards.

Path: ./contracts/Distributor.sol : updateAdminClaimPeriod(), adminClaim()

Recommendation: To make the change to *_adminClaimPeriod* not retroactive, a timelock should be put in place.

When *_adminClaimPeriod* gets updated, the new value should come into effect only after a time window as long as the old *_adminClaimPeriod* value has passed.

Found in: bbd2749

Status: New

Low

L01. Missing Feedback At Function removeFromWhiteList()

Impact	Low
Likelihood	Medium

The function *removeFromWhiteList()* does not provide any feedback regarding the success of the removal. The function should either return a bool false or revert in case of failure.

Path: ./contracts/Lingo.sol : removeFromWhiteList()

Recommendation: Let the function return a boolean false or revert if the user has not been found in the whitelist.

Found in: bbd2749

Status: New

L02. Inconsistent Metrics

Impact	Low
Likelihood	Medium

The mapping *_isUser* and the array *_userAddresses* are updated on user deposits, but are not cleared once users withdraw their entire positions

Path: ./contracts/Lingo.sol : deposit()

Recommendation: Update the *_isUser* and *_userAddresses* storage variables once the users withdraw their entire positions.

Found in: bbd2749

Status: New

L03. Missing SafeERC20 Library for Token Transfers

Impact	Medium

Likelihood	Low

The functions *deposit()* and *_transferTokens()* do not use SafeERC20 library for checking result of ERC20 token transfers. Tokens may not follow ERC20 standard and return false in case of transfer failure or not returning any value at all.

It is acknowledged that the SafeERC20 library is not required by Lingo token contract, but since the Lingo contract is not deployed by the Distribution contract in the constructor, it can't be assured that the Lingo contract in the audit scope will be tied to the *Distribution._token* variable, so SafeERC20 shall be implemented.

Path: ./contracts/Lingo.sol : deposit(), _transferTokens()

Recommendation: Implement the SafeERC20 library to interact with *_token* safely.

Found in: bbd2749

Status: New

L04. Unvalidated Constructor Parameters

Impact	Medium
Likelihood	Low

Constructor parameters are not validated, despite the contract implements setters with validation for such variables.

Paths: ./contracts/Distributor.sol : constructor()

Recommendation: Use the already existing validated setters to set the storage variables in the constructor.

Found in: bbd2749

Status: New

Informational

I01. Redundant ERC20 Import

It is not needed to import and inherit the ERC20 contract given that ERC20Burnable is already inherited.

Path: ./contracts/Lingo.sol : ERC20

Recommendation: Remove the redundant ERC20 import and inheritance.

Found in: bbd2749

Status: New

I02. Redundant Dynamic Array Creation

The *_setDefaultWhitelist()* function populates a static array and then converts it to a dynamic one to be passed to the *addToWhiteList()* function. The dynamic array can be directly built without double array initialization.

This affects the code readability.

Path: ./contracts/Lingo.sol : _setDefaultWhitelist()

Recommendation: It is recommended to replace second array initialization and cloning with straightforward array initialization:

```
address[] memory defaultWhiteListedAddresses = new address[](3);
defaultWhiteListedAddresses[0] = owner();
defaultWhiteListedAddresses[1] = address(this);
defaultWhiteListedAddresses[2] = _treasuryWallet;
```

Found in: bbd2749

Status: New

I03. Presence Of Magic Numbers

The contract has multiple “magic” numbers.

This affects the code readability and might cause issues during the further system development.

Paths: ./contracts/Lingo.sol

./contracts/Distribution.sol

Recommendation: Replace the magic numbers with constant values.

Found in: bbd2749

Status: New

I04. Redundant _msgSender(), Meta-Transactions Not Implemented

The *_msgSender()* function is needed to handle the meta transactions, in the OpenZeppelin library, it is used to support the development of the libraries which might be used with the contracts with the specified *TrustedForwarder* or to be used in such contracts directly.

However, the current implementation is not a library and does not rewrite the *_msgSender()* function to support meta transactions.

This affects the code clarity.

Paths: ./contracts/Distribution.sol

./contracts/Lingo.sol

Recommendation: Replace the *_msgSender()* with a regular *msg.sender*.

Found in: bbd2749

Status: New

I05. Duplicated Code

The mappings *_isInternalWhiteListed* and *_isExternalWhiteListed* can be merged into a single one by adding a key to distinguish between internal and external. The same applies for *_internalWhitelistedAddresses* and *_externalWhitelistedAddresses*.

This will allow to merge the functions *_removeFromInternalWhiteList()* and *_removeFromExternalWhiteList()*, as well as simplifying the logic of the function *addToWhiteList()*

Path: ./contracts/Lingo.sol : _removeFromInternalWhiteList(), _removeFromExternalWhiteList(), addToWhiteList()

Recommendation: Rework the mentioned functions to remove the code duplications.

Found in: bbd2749

Status: New

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/LingoCoin/lingo-app
Commit	bbd27496048abc36514ae5a927b3a8410f971e3c
Whitepaper	None
Requirements	File: Lingo Token Platform.pdf SHA3: 9284ee6124d2985443ccc474f40ff339c6ae3f6f146295ff1c27d9380187485b
Technical Requirements	File: Lingo - Technical Requirements.pdf SHA3: b78b962e4fbb356c2595e0cd2dcf11c28881989d4e15fac0878288a018d90fe0
Contracts	File : ./contracts/Distribution.sol SHA3: 53d12316bd341cce0c58a94efb2437943aa9791c7e7cab96ce3e767805cd1de4 File : ./contracts/Lingo.sol SHA3: 149f0ed83f4ec09dea9e2485858ae2909613e79dc180c7c9106c694f591a4024