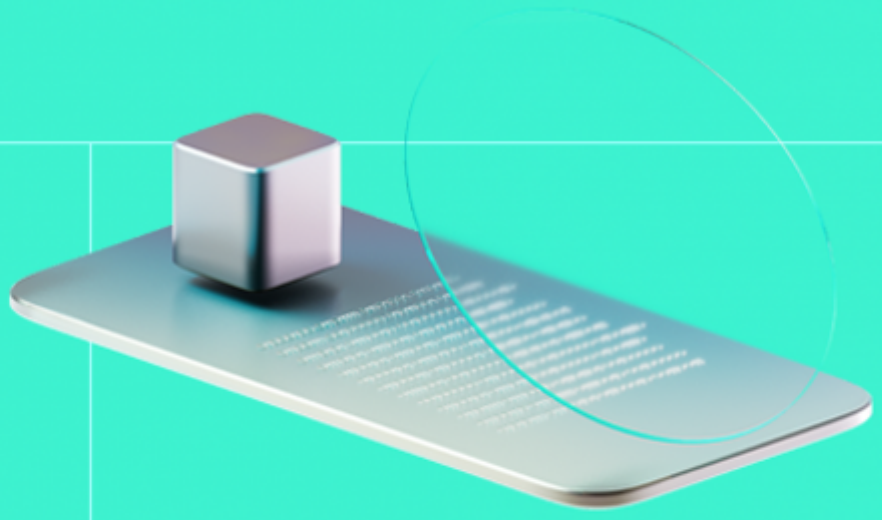
HACKEN

Smart Contract Code Review And Security Analysis Report

Customer: Linear

Date: 09/05/2024



We express our gratitude to the Linear team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

LiNEAR Protocol is a liquid staking solution built on the NEAR Protocol. LiNEAR unlocks liquidity of the staked NEAR by creating a staking derivative to be engaged with various DeFi protocols on NEAR and Aurora, while also enjoying over 10% APY staking rewards of the underlying base tokens. LiNEAR is the cornerstone piece of the NEAR-Aurora DeFi ecosystem.

Platform: Near

Language: Rust

Tags: Liquid Staking Protocol

Timeline: 22/03/2024 - 30/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/linear-protocol/LiNEAR
Commit	ccd1eeb9f2c2303c6ea2dd8072f833edb5e372e9

Audit Summary

10/10	10/10	-	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 10/10

The system users should acknowledge all the risks summed up in the risks section of the report

2	1	0	1
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	1
Low	1

Vulnerability	Status
F-2024-1979 - Lack of user charges for new deposit account creation leading to potential griefing attacks	Mitigated
F-2024-1983 - Inability to free allocated storage for unused accounts	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Linear
Audited By	Przemyslaw Swiatowiec
Approved By	Ataberk Yavuzer
Website	https://linearprotocol.org
Changelog	18/04/2024 - Preliminary Report
	30/04/2024 - Final Report
	09/05/2024 - Final Report Second Revision

Table of Contents

System Overview

LiNEAR Protocol is a non-custodial liquid staking protocol built on NEAR blockchain. Users can stake $NEAR via LiNEAR Protocol, receiving PoS staking rewards same as every other $NEAR staker, but also receive liquid $LiNEAR token which could be used in DeFi protocols. Furthermore, users can unstake their $LiNEAR and receive back $NEAR instantly with no waiting time, while a small portion of fees will be charged.

Base functionality provided for users:

Stake $NEAR
A user who wants to participate in NEAR staking can stake (deposit) his $NEAR to LiNEAR contract and get back $LiNEAR tokens. The $LiNEAR tokens are standard NEP141 tokens so that he can transfer/sell them freely.

Unstake $LiNEAR
If the user wants to exit staking and get his $NEAR back with his rewards together, he can unstake(withdraw) $NEAR by burning $LiNEAR tokens. There are two approaches he can take:

1. Instant Unstake - swap $LiNEAR tokens for $NEAR instantly from a liquidity pool in ref.finance. This way the user needs to bear transaction fees and slippage, however he can get $NEAR back immediately without any delay.
2. Delayed Unstake - if the user does not want to pay for any swap fees he can choose delayed unstake. Delayed unstake will make LiNEAR contract to actually unstake from underlying validators so that the exchange rate can always be guaranteed. There is a delay between initiating the unstake and actual receiving the $NEAR back. Typically it takes 4 epoches (~3 days).

Privileged roles

- Users can:
  - Deposit NEAR tokens
  - Stake NEAR tokens
  - Unstake NEAR tokens
  - Withdraw unstaked NEAR tokens
- Contract owner can:
  - add and remove managers
  - add and remove staking reward beneficiaries
  - set/modify treasury address
  - set address to contract with validator whitelist logic
  - pause and resume protocol
  - upgrade contract to new version
- Manager can:
  - add and remove whitelisted validators and assign validator weight
  - update validator's base amounts
  - sync validators balance with LiNear contract
  - run drain unstake and drain withdraw operations

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

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

Code quality

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

Test coverage

Code coverage of the project was not provided as there is no reliable tool to verify code coverage of Near smart contracts.

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **1** medium, and **1** low severity issues. The high issue was mitigated and the low was fixed in the remediation part of an audit process, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **10**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

Risk Statement

This audit report focuses exclusively on the security assessment of the contracts within the specified review scope. Interactions with out-of-scope contracts are presumed to be correct and are not examined in this audit. We want to highlight that Interactions with contracts outside the specified scope have not been verified or assessed as part of this report. This includes staking contracts and validator whitelisting contract.

While we have diligently identified and mitigated potential security risks within the defined scope, it is important to note that our assessment is confined to the isolated contracts within this scope. The overall security of the entire system, including external contracts and integrations beyond our audit scope, cannot be guaranteed.

Users and stakeholders are urged to exercise caution when assessing the security of the broader ecosystem and interactions with external contracts. For a comprehensive evaluation of the entire system, additional audits and assessments outside the scope of this report are necessary.

This report serves as a snapshot of the security status of the audited contracts within the specified scope at the time of the audit. We strongly recommend ongoing security evaluations and continuous monitoring to maintain and enhance the overall system's security.

Permissions, Authorization & Access

Insufficient Permissioning and Documentation: Inadequate permissions and under-documentation heighten the risk of unauthorized access and actions, complicating issue resolution and increasing the potential for security breaches.

Absence of Time-lock Mechanisms for Critical Operations: Without time-locks on critical operations, there is no buffer to review or revert potentially harmful actions, increasing the risk of rapid exploitation and irreversible changes.

Centralization

Administrative Key Control Risks: The digital contract architecture relies on administrative keys for critical operations. Centralized control over these keys presents a significant security risk, as compromise or misuse can lead to unauthorized actions or loss of funds.

Single Entity Upgrade Authority: The contract grants a single entity the authority to implement upgrades or changes. This centralization of power risks unilateral decisions that may not align with the community or stakeholders' interests, undermining trust and security.

Upgradeability

Flexibility and Risk in Contract Upgrades: The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.

Absence of Upgrade Window Constraints: The contract suite allows for immediate upgrades without a mandatory review or waiting period, increasing the risk of rapid deployment of malicious or flawed code, potentially compromising the system's integrity and user assets.

Findings

Vulnerability Details

F-2024-1979 - Lack of user charges for new deposit account creation leading to potential griefing attacks - Medium

Description:

It was observed that the contract does not charge users for the storage used when creating new deposit accounts (`deposit` function). In the NEAR protocol, the contract's balance is utilized to pay for storage costs. This oversight allows users, including potential malicious actors, to create deposit accounts with zero token deposits, leading to the allocation of new storage slots at the contract's expense.

Our tests revealed that the cost for adding a new `Account` struct to storage is approximately `0.004 NEAR`, while the transaction cost paid by a user is only `0.0005 NEAR`. This discrepancy indicates that the attacker can lock contract funds by utilizing storage slots creation for new accounts with low cost.

This situation is exacerbated by two facts:

1. Currently, zero-amount deposits are permitted.
2. The contract lacks a mechanism to reclaim or free up storage space once allocated.

This can lead to a scenario where a malicious actor deliberately drains the contract's available balance by creating numerous empty deposit accounts. This way there would be no funds left in the contract to accept new deposits - the contract is going to miss Near to pay for new storage.

What is more, in the current implementation users deposit tokens and do not stake them. Such deposited tokens are part of the contract balance. As a result, malicious actor can lock not only funds that were added by protocol owners but also can potentially lock legitimate user deposits within the contract without the possibility of withdrawal or utilization.

Affected code:

```
#[payable]
pub fn deposit(&mut self) {
    let amount = env::attached_deposit();
    self.internal_deposit(amount);
}

pub(crate) fn internal_deposit(&mut self, amount: Balance) {
    self.assert_running();

    let account_id = env::predecessor_account_id();
    let mut account = self.internal_get_account(&account_id);
    account.unstaked += amount;
    self.internal_save_account(&account_id, &account);

    Event::Deposit {
        account_id: &account_id,
```

```
            amount: &U128(amount),
            new_unstaked_balance: &U128(account.unstaked),
        }
        .emit();
    }

    #[payable]
    fn storage_withdraw(&mut self, amount: Option<U128>) -> StorageBalance {
        assert_one_yocto();
        let predecessor_account_id = env::predecessor_account_id();
        if let Some(storage_balance) = self.internal_storage_balance_of(&predecessor_account_id
            match amount {
                Some(amount) if amount.0 > 0 => {
                    env::panic_str("The amount is greater than the available storage balance");
                }
                _ => storage_balance,
            }
        } else {
            env::panic_str(
                format!("The account {} is not registered", &predecessor_account_id).as_str(),
            );
        }
    }

    /// Storage unregister is disabled because staking users don't need
    /// to deposit but they are allowed to withdraw storage fee with
    /// the current implementation.
    #[payable]
    fn storage_unregister(&mut self, force: Option<bool>) -> bool {
        panic!("Storage unregister is not supported yet.");
    }
```

Status: Mitigated

Classification

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 3
Exploitability [0-2]: 0
Complexity [0-2]: 2
Final Score: 2.6 (Medium)
Hacken Calculator Version: 0.6

Severity: Medium

Recommendations

Remediation: To mitigate the risks associated with free storage allocation for new deposit accounts and protect the contract's balance from potential griefing attacks, it is recommended that the contract implement a charging mechanism for storage slots used by users. Users should bear the cost of the storage required for their deposit accounts to prevent malicious exploitation and ensure the contract's sustainability.

Remediation (commit id: aa045346b87feca978476121883ab1611b487600): the deposit and stake interfaces are designed to align with the staking pool core contract, thereby ensuring consistency across the platform. Given the infrequency of griefing attacks and the impact on user experience and cross-contract interactions, the decision to forego storage fees is deemed appropriate. The strategy to disallow zero-amount deposits has been adopted to mitigate potential risks while maintaining seamless integration and functionality.

Evidences

PoC

Reproduce:

```
test(
  'Drain contract tokens',
  async (t) => {
    const { contract, alice, root, owner } = t.context;
    const whitelist = await root.createSubAccount("whitelist2");
    await whitelist.deploy("compiled-contracts/mock_whitelist.wasm");
    await root.call(whitelist, 'new', {});
    const validator1 = await root.createSubAccount('validator1', {
      initialBalance: NEAR.parse('1000000').toString(),
    });
    await root.call(whitelist, 'add_whitelist', {"account_id": validator1.accountId});

    await owner.call(contract, 'set_whitelist_contract_id', {
      account_id: whitelist.accountId,
    });

    let drained = false;
    let counter = 0;
    // legit user deposits
    const legitDeposit = NEAR.parse('1');
    await alice.call(contract, 'deposit', {}, { attachedDeposit: legitDeposit });
    while(!drained) {
      counter++;
      try {
        const user = await root.createSubAccount(counter.toString(), {
          initialBalance: NEAR.parse('1000000').toString(),
        });
        const deposit = NEAR.parse('0');
        await user.call(contract, 'deposit', {}, { attachedDeposit: deposit });
        const contractBalance = await contract.balance();
        console.log(contractBalance.total.toHuman());
        const userBalance = user.balance();
        console.log((await userBalance).total.toHuman());
        break;
      } catch(e) {
        console.log(e);
        drained = true;
      }
    }
```

```
// observe that contract balance is 0 and user cannot create new deposits
const contractBalance = await contract.balance();
console.log(contractBalance.total.toHuman());
await alice.call(contract, 'deposit', {}, { attachedDeposit: legitDeposit });
}
```

F-2024-1983 - Inability to free allocated storage for unused accounts - Low

Description:

During the security and functionality review of the smart contract, it was observed that there is no mechanism in place to free up storage once it has been allocated. This limitation is particularly problematic for accounts that are no longer in use. In the context of the NEAR protocol, where contracts are charged for storage allocation, the inability to reclaim storage space for unused accounts can lead to inefficient use of resources and unnecessary financial costs for the users. The loss primarily impacts users who send LiNEAR tokens to recipients without previous deposits or stakes in the protocol, as they must cover a non-refundable storage fee, which neither the sender nor the recipient can retrieve.

Affected code:

```
#[payable]
fn storage_withdraw(&mut self, amount: Option<U128>) -> StorageBalance {
    assert_one_yocto();
    let predecessor_account_id = env::predecessor_account_id();
    if let Some(storage_balance) = self.internal_storage_balance_of(&predecessor_account_id
        match amount {
            Some(amount) if amount.0 > 0 => {
                env::panic_str("The amount is greater than the available storage balance");
            }
            _ => storage_balance,
        }
    } else {
        env::panic_str(
            format!("The account {} is not registered", &predecessor_account_id).as_str(),
        );
    }
}

/// Storage unregister is disabled because staking users don't need
/// to deposit but they are allowed to withdraw storage fee with
/// the current implementation.
#[payable]
fn storage_unregister(&mut self, force: Option<bool>) -> bool {
    panic!("Storage unregister is not supported yet.");
}
```

Status: Fixed

Classification

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 2
Exploitability [0-2]: 0
Complexity [0-2]: 2
Final Score: 2.1 (Low)
Hacken Calculator Version: 0.6

Severity: Low

Recommendations

Remediation: To address the identified issue and improve the contract's efficiency and cost-effectiveness, it is recommended to implement a mechanism that allows for the freeing of storage for accounts that are no longer in use. This would involve creating a method within the contract that can be invoked to delete unused accounts and reclaim the associated storage space.

Remediation (commit Id: aa045346b87feca978476121883ab1611b487600): the proposal to introduce a mechanism for freeing storage of inactive accounts is reconsidered in light of F-2024-1979, which effectively addresses related concerns. Due to potential security risks highlighted by the client, such as attackers exploiting minimal deposits for disproportionate withdrawals, it is recommended to continue prohibiting storage withdrawals and account unregistering. This approach ensures the ongoing integrity and security of the contract as established by F-2024-1979.

Observation Details

F-2024-1975 - Lack of event emissions for critical state changes - Info

Description:

The `LiNear` contract does not emit events for critical actions and state changes, particularly in functions such as `set_owner`, `add_manager`, `remove_manager`, `set_beneficiary`, `remove_beneficiary`, `set_treasury`, `set_whitelist_contract_id`, `pause`, `resume`. Events in Near contracts are crucial for tracking changes and actions off-chain, providing transparency and auditability for contract interactions. The absence of events for these critical operations means that external systems, interfaces, and users lack a reliable, on-chain method to detect and respond to these significant changes. This oversight can impede monitoring, analytics, and integration with off-chain systems, ultimately affecting the contract's usability and governance.

Affected code:

```
pub fn set_owner(&mut self, new_owner_id: AccountId) {
    self.assert_owner();
    self.owner_id = new_owner_id;
}

pub fn add_manager(&mut self, new_manager_id: AccountId) {
    self.assert_running();
    self.assert_owner();
    self.internal_add_manager(&new_manager_id);
}

pub fn remove_manager(&mut self, manager_id: AccountId) -> bool {
    self.assert_running();
    self.assert_owner();
    self.internal_remove_manager(&manager_id)
}

pub fn set_beneficiary(&mut self, account_id: AccountId, bps: u32) {
    ...
}

pub fn remove_beneficiary(&mut self, account_id: AccountId) {
    self.assert_running();
    self.assert_owner();
    //@audit-info no events on critical actions
    self.beneficiaries.remove(&account_id);
}

/// Set account ID of the treasury
pub fn set_treasury(&mut self, account_id: AccountId) {
    //some actions can be blocked
    self.assert_running();
    self.assert_owner();
    self.treasury_id = account_id;
}
```

```rust
/// Set whitelist account ID
pub fn set_whitelist_contract_id(&mut self, account_id: AccountId) {
    self.assert_running();
    self.assert_owner();
    self.whitelist_account_id = Some(account_id);
}

// --- Pause ---

pub fn pause(&mut self) {
    self.assert_owner();
    require!(!self.paused, ERR_ALREADY_PAUSED);
    self.paused = true;
}

pub fn resume(&mut self) {
    self.assert_owner();
    require!(self.paused, ERR_NOT_PAUSED);
    self.paused = false;
}
```

Status: Fixed

Recommendations

Remediation: Modify the contract to emit specific events for all critical state changes.

Remediation (commit id: ee3088aa7abe62e571edabbab15c2774d1416b94): events for the aforementioned changes are emitted.

F-2024-1977 - Unnecessary double checks in add_manager and remove_manager functions - Info

Description:

During the security audit of the `LiNear` contract, it was observed that the functions `add_manager` and `remove_manager` contain redundant state checks for the contract's pause status (`assert_running` function), The `assert_running` function which validates that the contract is not paused, is invoked twice within the same transactional context for each of the aforementioned functions. This redundancy results in unnecessary computation, thereby increasing the execution cost for users performing these operations.

The `assert_running` is verified both in `add_manager`/`remove_manager` and `internal_add_manager`/`internal_remove_manager`:

```
pub fn add_manager(&mut self, new_manager_id: AccountId) {
    self.assert_running();
    self.assert_owner();
    self.internal_add_manager(&new_manager_id);
}

pub fn remove_manager(&mut self, manager_id: AccountId) -> bool {
    self.assert_running();
    self.assert_owner();
    self.internal_remove_manager(&manager_id)
}

pub(crate) fn internal_add_manager(&mut self, manager_id: &AccountId) {
    self.assert_running();
    self.managers.insert(manager_id);
}

pub(crate) fn internal_remove_manager(&mut self, manager_id: &AccountId) -> bool {
    self.assert_running();
    self.managers.remove(manager_id)
}
```

What is more, the managers struct updates are also happening in contract initialization - the contract owner is automatically set as the manager. During initialization, there is no need to check for pause status as `pause` variable is set to `false` in the initialization function.

```
let mut this = Self {
    owner_id: owner_id.clone(),
    managers: UnorderedSet::new(StorageKey::Managers),
    treasury_id: owner_id.clone(),
    total_share_amount: 10 * ONE_NEAR,
    total_staked_near_amount: 10 * ONE_NEAR,
    accounts: UnorderedMap::new(StorageKey::Accounts),
    paused: false,
    account_storage_usage: 0,
    beneficiaries: UnorderedMap::new(StorageKey::Beneficiaries),
    // Validator Pool
```

```
    validator_pool: ValidatorPool::new(),
    whitelist_account_id: None,
    epoch_requested_stake_amount: 10 * ONE_NEAR,
    epoch_requested_unstake_amount: 0,
    stake_amount_to_settle: 0,
    unstake_amount_to_settle: 0,
    last_settlement_epoch: 0,
};
this.internal_add_manager(&owner_id);
this.measure_account_storage_usage();
this
```

Status: Fixed

Recommendations

Remediation: It is recommended to check for contract pause state only once - either in internal or external functions.

Remediation (commit id: 4e8a6d294aaaa9d5fee6fc154f2ba26fa8a2a50d): the double checks were removed.

F-2024-1978 - Lack of two-step ownership transfer pattern - Info

Description: The current implementation of the `LiNear` smart contract allows for the immediate transfer of contract ownership through a single transaction, without an intermediate confirmation step. This one-step ownership transfer mechanism permits the current owner to set any address as the new owner directly, without a secondary verification or acceptance step from the new owner. This approach exposes the contract to risks associated with locking contract ownership if an incorrect new owner is set. Such a contract could be unlocked only by upgrading the contract to a new version (migration process).

```
pub fn set_owner(&mut self, new_owner_id: AccountId) {
    self.assert_owner();
    self.owner_id = new_owner_id;
}
```

Status: Mitigated

Recommendations

Remediation: Consider implementing a two-step ownership transfer pattern.

Remediation: the client states that since all management activities, including changes in ownership, are handled by the DAO, implementing a two-step ownership transition pattern is redundant.

F-2024-2164 - Inconsistent account data retrieval in internal_get_account function - Info

Description:

It was observed that the `internal_get_account` function within the smart contract inconsistently retrieves account data due to the use of `unwrap_or_default`. This method returns a default account structure when an account does not exist, instead of throwing an error.

As a result, this improper error handling causes downstream inaccuracies in related view functions such as `get_account_details` and `get_account`. These functions incorrectly report that non-existent accounts can withdraw funds (`can_withdraw` set to `true`) and display other account values as **0**. This behavior can mislead users or external systems interacting with the contract by presenting inaccurate account status and data.

```
pub fn get_account(&self, account_id: AccountId) -> HumanReadableAccount {
    let account = self.internal_get_account(&account_id);
    HumanReadableAccount {
        account_id,
        unstaked_balance: account.unstaked.into(),
        staked_balance: self
            .staked_amount_from_num_shares_rounded_down(account.stake_shares)
            .into(),
        can_withdraw: account.unstaked_available_epoch_height <= get_epoch_height(),
    }
}

pub fn get_account_details(&self, account_id: AccountId) -> AccountDetailsView {
    let account = self.internal_get_account(&account_id);
    AccountDetailsView {
        account_id,
        unstaked_balance: account.unstaked.into(),
        staked_balance: self
            .staked_amount_from_num_shares_rounded_down(account.stake_shares)
            .into(),
        unstaked_available_epoch_height: account.unstaked_available_epoch_height,
        can_withdraw: account.unstaked_available_epoch_height <= get_epoch_height(),
    }
}

pub(crate) fn internal_get_account(&self, account_id: &AccountId) -> Account {
    self.accounts.get(account_id).unwrap_or_default()
}
```

Status: Accepted

Recommendations

Remediation:

To correct data inconsistencies and improve the integrity of account information retrieved from the smart contract, it is recommended to modify the

`internal_get_account` function to throw an appropriate error when an account does not exist, instead of returning a default account structure.

Remediation: to maintain compatibility with existing modules, the client has decided not to alter the `get_account` function. This decision ensures consistency with the implementation of the staking pool contract.

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.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
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 will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

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

Scope Details

Repository	https://github.com/linear-protocol/LiNEAR
Commit	ccd1eeb9f2c2303c6ea2dd8072f833edb5e372e9
Whitepaper	https://docs.linearprotocol.org/developer-guide/contract-overview
Requirements	https://docs.linearprotocol.org/developer-guide/contract-overview
Technical Requirements	https://docs.linearprotocol.org/developer-guide/contract-overview

Contracts in Scope

contracts/linear/*