BlockSec

Security Audit Report for LiNEAR

Date: Apr 1st, 2022

Version: 1.0

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	LiNEAR Protocol
Target	LiNEAR

Version History

Version	Date	Description
1.0	Apr 1st, 2022	First Release

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The repository that has been audited includes LiNEAR [1].

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project	Version	Commit SHA
LiNEAR	`Version 1`	`fdbacc68c98205cba9f42c130d464ab3114257b6`
	`Version 2`	`41bb61f72a7575861b39faa5e888a86148958c5f`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

[1] https://github.com/linear-protocol/LiNEAR

- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find 4 potential issues in the smart contract. We also have 4 recommendations, as follows:

- High Risk: 0
- Medium Risk: 2
- Low Risk: 2
- Recommendations: 4

ID	Severity	Description	Category	Status
1	Medium	*Precision loss*	Software Security	Fixed
2	Low	*User's available balance may be locked temporarily*	DeFi Security	Confirmed
3	Medium	*Unlimited reward distribution to beneficiaries*	DeFi Security	Fixed
4	Low	*Users' unstaking requests may not be satisfied in time*	DeFi Security	Fixed
7	-	*Function* `epoch_update_rewards` *may not work due to unlimited beneficiaries*	Recommendation	Fixed
8	-	*Redundant code*	Recommendation	Confirmed
6	-	*Missing check on the* `prepaid_gas` *in function* `ft_transfer_call`	Recommendation	Fixed
9	-	*The risk of centralized design*	Recommendation	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Precision loss

Status Fixed in `version 2`

Introduced by `version 1`

Description In line 125 of function `internal_calculate_distribution`, division is performed before multiplication when calculating variable `reward_per_session` .

```
110    fn internal_calculate_distribution(
111        &self,
112        farm: &Farm,
113        total_staked: Balance,
114    ) -> Option<RewardDistribution> {
115        if farm.start_date > env::block_timestamp() {
116            // Farm hasn't started.
117            return None;
118        }
119        let mut distribution = farm.last_distribution.clone();
120        if distribution.undistributed == 0 {
121            // Farm has ended.
122            return Some(distribution);
123        }
124        distribution.reward_round = (env::block_timestamp() - farm.start_date) / SESSION_INTERVAL;
```

```
    let reward_per_session =
        farm.amount / (farm.end_date - farm.start_date) as u128 * SESSION_INTERVAL as u128;
```

Listing 2.1: contracts/linear/src/farm.rs

Impact Division truncates for integers in Rust language. In this case, division before multiplication for intergers may result in precision loss.

Suggestion I Modify this calculation to perform multiplication before division.

Suggestion II Pre-calculate the value of `reward_per_session` when a farm is added. This is because `farm.amount`, `farm.end_date` and `farm.start_date` do not change during farming process unless the owner stops it.

2.2 DeFi Security

2.2.1 User's available balance may be locked temporarily

Status Confirmed

Introduced by `version 1`

Description User's deposited `NEARs` will be added into the user's unstaked balance directly. Therefore, if the user invoked the `unstake/unstake_all` functions, that amount of available `NEARs` will be locked in the next 0 to 8 epochs.

```
pub(crate) fn internal_deposit(&mut self, amount: Balance) {
  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();
}
```

Listing 2.2: contracts/linear/src/internal.rs

Impact If a user is unaware of this contract workflow and directly interacts with this contract, the user's available balance may be locked temporarily.

Suggestion I Add another attribute (e.g., `available_amount`) to struct `Account` to maintain the available `NEARs`.

Feedback from the Project This is by design, basically following the original interface and design of the staking pool. To resolve the potential issue, we can make enhancements to add another field 'unstaking' to distinguish from 'unstaked', and move 'unstaking' into 'unstaked' before we kick off the next unstaking process for this account. But for now, we'd prefer to not make the change at the moment, to keep the workflow consistent with staking pool. As a workaround, when users are unstaking from frontend, the UI will remind users to withdraw first if they have 'unstaked' amount in their account.

2.2.2 Unlimited reward distribution to beneficiaries

Status Fixed in `version 2`

Introduced by `version 1`

Description This contract does not check the total weight of all beneficiaries in the function `assert_valid` when setting up a new beneficiary.

```rust
pub fn set_beneficiary(&mut self, account_id: AccountId, fraction: Fraction) {
  self.assert_owner();
  fraction.assert_valid();
  self.beneficiaries.insert(&account_id, &fraction);
}
```

Listing 2.3: contracts/linear/src/owner.rs

```rust
pub fn assert_valid(&self) {
  require!(self.denominator != 0, ERR_FRACTION_BAD_DENOMINATOR);
  require!(
    self.numerator <= self.denominator,
    ERR_FRACTION_BAD_NUMERATOR
  );
}
```

Listing 2.4: contracts/linear/src/utils.rs

Impact Once the total weight of beneficiaries exceeds 100%, the `LiNEARs` minted for beneficiaries may decrease the price of `LiNEAR` after the execution of action `epoch_update_rewards`.

Suggestion I Introduce a reasonable threshold to limit the total reward distributed to beneficiaries.

2.2.3 Users' unstaking requests may not be satisfied in time

Status Fixed in `version 2`

Introduced by `version 1`

Description The number of epochs returned from function `get_num_epoch_to_unstake` should be doubled in some corner cases. For example, if the total `NEARs` staked in validator staking pools, which are not in pending status, is not enough, the users can not withdraw all the unstaked `NEARs` requested after 4 epochs.

```rust
pub fn get_num_epoch_to_unstake(&self, _amount: u128) -> EpochHeight {
  // the num of epoches can be doubled or trippled if not enough stake is available
  NUM_EPOCHS_TO_UNLOCK
}
```

Listing 2.5: contracts/linear/src/validator_pool.rs

Impact User's unstaking requests may not always be satisfied in time.

Suggestion I Implement a strategy to predict the user's unstaking waiting time based on validator staking pools' status.

2.3 Additional Recommendation

2.3.1 Function `epoch_update_rewards` may not work due to unlimited beneficiaries

Status Fixed in `version 2`

Introduced by `version 1`

Description The number of beneficiaries is unlimited. In this case, function `epoch_update_rewards` that invokes function `internal_distribute_staking_rewards` may run out of gas.

```rust
pub fn epoch_update_rewards(&mut self, validator_id: AccountId) {
  let min_gas = GAS_EPOCH_UPDATE_REWARDS + GAS_EXT_GET_BALANCE + GAS_CB_VALIDATOR_GET_BALANCE;
  require!(
    env::prepaid_gas() >= min_gas,
    format!("{}. require at least {:?}", ERR_NO_ENOUGH_GAS, min_gas)
  );

  let validator = self
    .validator_pool
    .get_validator(&validator_id)
    .expect(ERR_VALIDATOR_NOT_EXIST);

  if validator.staked_amount == 0 && validator.unstaked_amount == 0 {
    return;
  }

  validator
    .refresh_total_balance()
    .then(ext_self_action_cb::validator_get_balance_callback(
      validator.account_id,
      env::current_account_id(),
      NO_DEPOSIT,
      GAS_CB_VALIDATOR_GET_BALANCE,
    ));
}
```

Listing 2.6: contracts/linear/src/epoch_actions.rs

```rust
  /// When there are rewards, a part of them will be
  /// given to executor, manager or treasury by minting new LiNEAR tokens.
  pub(crate) fn internal_distribute_staking_rewards(&mut self, rewards: Balance) {
    let hashmap: HashMap<AccountId, Fraction> = self.internal_get_beneficiaries();
    for (account_id, fraction) in hashmap.iter() {
      let reward_near_amount: Balance = fraction.multiply(rewards);
      // mint extra LiNEAR for him
      self.internal_mint_beneficiary_rewards(&account_id, reward_near_amount);
    }
  }
```

Listing 2.7: contract/src/internal.rs

Impact Function `epoch_update_rewards` may not work due to limited gas when there are too many beneficiaries.

Suggestion I It is recommended to add a reasonable threshold to limit the number of beneficiaries.

2.3.2 Redundant code

Status Confirmed
Introduced by `version 1`

Description Parameter `registration_only` is redundant as function `storage_deposit` do not implement any logic for this parameter.

```rust
fn storage_deposit(
  &mut self,
  account_id: Option<AccountId>,
  registration_only: Option<bool>,
) -> StorageBalance {
  let amount: Balance = env::attached_deposit();
  let account_id = account_id.unwrap_or_else(env::predecessor_account_id);
  if let Some(account) = self.accounts.get(&account_id) {
    log!("The account is already registered, refunding the deposit");
    if amount > 0 {
      Promise::new(env::predecessor_account_id()).transfer(amount);
    }
  } else {
    let min_balance = self.storage_balance_bounds().min.0;
    if amount < min_balance {
      env::panic_str("The attached deposit is less than the minimum storage balance");
    }

    self.internal_register_account(&account_id);
    let refund = amount - min_balance;
    if refund > 0 {
      Promise::new(env::predecessor_account_id()).transfer(refund);
    }
  }
  self.internal_storage_balance_of(&account_id).unwrap()
}
```

Listing 2.8: contracts/linear/src/fungible_token/storage.rs

Suggestion I It is recommended to remove this unused parameter in function `storage_deposit`.

Feedback from the Project That's true. The same happens to the standard FT implementation in near-contract-standards crate. We'll keep the unused `registration_only` parameter to keep consistent with the standard `storage_deposit(account_id, registration_only)` interface.

2.3.3 Missing check on the `prepaid_gas` in function `ft_transfer_call`

Status Fixed in `version 2`
Introduced by `version 1`

Description The `prepaid_gas` should be checked to ensure it is enough for the target functions including `ft_on_transfer` and `ft_resolve_transfer`.

```rust
#[payable]
fn ft_transfer_call(
  &mut self,
  receiver_id: AccountId,
  amount: U128,
  memo: Option<String>,
  msg: String,
) -> PromiseOrValue<U128> {
  assert_one_yocto();
  let sender_id = env::predecessor_account_id();
  let amount = amount.into();
  self.internal_ft_transfer(&sender_id, &receiver_id, amount, memo);
  // Initiating receiver's call and the callback
  ext_fungible_token_receiver::ft_on_transfer(
    sender_id.clone(),
    amount.into(),
    msg,
    receiver_id.clone(),
    NO_DEPOSIT,
    env::prepaid_gas() - GAS_FOR_FT_TRANSFER_CALL,
  )
  .then(ext_ft_self::ft_resolve_transfer(
    sender_id,
    receiver_id,
    amount.into(),
    env::current_account_id(),
    NO_DEPOSIT,
    GAS_FOR_RESOLVE_TRANSFER,
  ))
  .into()
}
```

Listing 2.9: contracts/linear/src/fungible_token/core.rs

Suggestion I Check the `prepaid_gas`.

2.3.4 The risk of centralized design

Status Confirmed
Introduced by `version 1`

Description This project has potential centralization problems.

Suggestion I It is recommended to introduce a decentralization design in the contract, such as multi-signature or DAO.

Feedback from the Project I Yes. This in plan as mentioned in github.com/linear-protocol/LiNEAR/issues/60

Suggestion II The project owner needs to ensure the security of the private key of the `OWNER_ROLE/MANAGERS_ROLE` and use a multi-signature scheme to reduce the risk of single-point failure.

Feedback from the Project II Yes. We have been working on security policies to reduce the risks of single point of failure.