Security Audit Report for Ref-ve

Date: July 14, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2022.07.25
15:19:50 +08'00'

Contents

1 Introduction **1**

1.1 About Target Contracts 1

1.2 Disclaimer 1

1.3 Procedure of Auditing 2

1.3.1 Software Security 2

1.3.2 DeFi Security 2

1.3.3 NFT Security 3

1.3.4 Additional Recommendation 3

1.4 Security Model 3

2 Findings **5**

2.1 Software Security 5

2.1.1 Unlimited Account Registeration without Storage Fees 5

2.1.2 Unlimited Length of Proposal.description 7

2.2 DeFi Security 8

2.2.1 User's Reward may be Lost 8

2.2.2 Unreasonable Duration of Proposal 9

2.3 Additional Recommendation 11

2.3.1 Unused Function 11

2.3.2 Lack of Checking on the Locking Duration 11

2.3.3 Lack of assert_one_yocto() 12

2.3.4 Lack of assert_one_yocto() 13

2.3.5 Lack of Checking on the Gas Used by migrate 14

2.3.6 Potential Centralization Problem 15

2.3.7 Potential Elastic Supply Token Problem 15

2.4 Notes 15

2.4.1 Action::VoteNonsense is invalid 15

Report Manifest

Item	Description
Client	Ref Finance
Target	Ref-ve

Version History

Version	Date	Description
1.0	July 14, 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 ref-ve [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 only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
ref-ve	`Version 1`	`1fd6dfe2160590bab0f8e9ccf17c4dcce2c42f33`
	`Version 2`	`87491b5eb55909f98ed3152fedaa5a65592d779f`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **ref-ve** folder contract only. Specifically, the files covered in this audit include:

- src/account.rs
- src/actions_of_account.rs
- src/actions_of_proposal.rs
- src/actions_of_reward.rs
- src/errors.rs
- src/events.rs
- src/lib.rs
- src/management.rs
- src/owner.rs
- src/proposals_action.rs
- src/proposals_incentive.rs
- src/proposals.rs
- src/storage_impl.rs
- src/token_receiver.rs
- src/utils.rs
- src/views.rs

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

[1] https://github.com/ref-finance/ref-ve

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.
- **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

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

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

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 item will fall into one of the following four categories:

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

Chapter 2 Findings

In total, we find **four** potential issues. We have **seven** recommendations and **one** note.

- High Risk: 0
- Medium Risk: 2
- Low Risk: 2
- Recommendations: 7
- Notes: 1

ID	Severity	Description	Category	Status
1	Medium	Unlimited Account Registeration without Storage Fees	Software Security	Fixed
2	Medium	Unlimited Length of Proposal.description	Software Security	Fixed
3	Low	User's Reward may be Lost	DeFi Security	Fixed
4	Low	Unreasonable Duration of Proposal	DeFi Security	Fixed
5	-	Unused Function	Recommendation	Fixed
6	-	Lack of Checking on the Locking Duration	Recommendation	Fixed
7	-	Lack of assert_one_yocto()	Recommendation	Fixed
8	-	Lack of assert_one_yocto()	Recommendation	Fixed
9	-	Lack of Checking on the Gas Used by migrate	Recommendation	Fixed
10	-	Potential Centralization Problem	Recommendation	Confirmed
11	-	Potential Elastic Supply Token Problem	Recommendation	Confirmed
12	-	Action::VoteNonsense is Invalid	Note	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Unlimited Account Registeration without Storage Fees

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Account can be registered with function `internal_unwrap_or_default_account` in function `lock_lpt`(line 101) or `append_lpt`(line 132) without a storage deposit. Meanwhile, `mft_on_transfer` does not limit the locking amount, which allows an account registration with very small amount of (e.g., 1 yocto) LP tokens locked.

```
 95    pub fn lock_lpt(
 96        &mut self,
 97        account_id: &AccountId,
 98        amount: Balance,
 99        duration_sec: u32,
100    ) {
101        let mut account = self.internal_unwrap_or_default_account(account_id);
102        let config = self.internal_config();
103        require!(duration_sec >= config.min_locking_duration_sec, E302_INVALID_DURATION);
104        require!(duration_sec <= config.max_locking_duration_sec, E302_INVALID_DURATION);
```

```

        let increased_ve_lpt = account.lock_lpt(amount, duration_sec, &config, self.data().
            lptoken_decimals);
        require!(increased_ve_lpt > 0, E101_INSUFFICIENT_BALANCE);
        self.mint_love_token(account_id, increased_ve_lpt);

        self.data_mut().cur_lock_lpt += amount;
        self.data_mut().cur_total_ve_lpt += increased_ve_lpt;

        self.update_impacted_proposals(&mut account, increased_ve_lpt, true);

        self.internal_set_account(account_id, account);

        Event::LptLock {
            caller_id: account_id,
            deposit_amount: &U128(amount),
            increased_ve_lpt: &U128(increased_ve_lpt),
            duration: duration_sec,
        }
        .emit();
    }

    pub fn append_lpt(
        &mut self,
        account_id: &AccountId,
        amount: Balance,
        append_duration_sec: u32,
    ) {
        let mut account = self.internal_unwrap_or_default_account(account_id);
        require!(account.unlock_timestamp != 0, E105_ACC_NOT_LOCKED);
        let timestamp = env::block_timestamp();
        let duration_sec = nano_to_sec(account.unlock_timestamp) - nano_to_sec(timestamp) +
            append_duration_sec;

        let config = self.internal_config();
        require!(duration_sec >= config.min_locking_duration_sec, E302_INVALID_DURATION);
        require!(duration_sec <= config.max_locking_duration_sec, E302_INVALID_DURATION);

        let increased_ve_lpt = account.lock_lpt(amount, duration_sec, &config, self.data().
            lptoken_decimals);
        require!(increased_ve_lpt > 0, E101_INSUFFICIENT_BALANCE);
        self.mint_love_token(account_id, increased_ve_lpt);

        self.data_mut().cur_lock_lpt += amount;
        self.data_mut().cur_total_ve_lpt += increased_ve_lpt;

        self.update_impacted_proposals(&mut account, increased_ve_lpt, true);

        self.internal_set_account(account_id, account);

        Event::LptAppend {
            caller_id: account_id,
            deposit_amount: &U128(amount),
            increased_ve_lpt: &U128(increased_ve_lpt),
            duration: duration_sec,
        }
        .emit();
    }
```

Listing 2.1: contracts/ref-ve/src/token_receiver.rs

Impact The contract is vulnerable to DoS attack. Malicious users can run out of storage by registering numerous users with function `lock_lpt`.

Suggestion I Change `internal_unwrap_or_default_account` to `internal_unwrap_account` to make sure the users are registered before locking/appending lpt.

Suggestion II Limit the minimum locking amount in function `mft_on_transfer`.

2.1.2 Unlimited Length of Proposal.description

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is no check on the length of `Proposal.description` when creating proposals.

```
#[payable]
pub fn create_proposal(
    &mut self,
    kind: ProposalKind,
    description: String,
    start_at: u32,
    duration_sec: u32,
) -> u32 {
    let proposer = env::predecessor_account_id();
    require!(self.data().whitelisted_accounts.contains(&proposer) , E002_NOT_ALLOWED);

    self.internal_unwrap_account(&proposer);

    let config = self.internal_config();

    require!(start_at - nano_to_sec(env::block_timestamp()) >= config.
        min_proposal_start_vote_offset_sec, E402_INVALID_START_TIME);

    let votes: Vec<VoteInfo> = match &kind {
        ProposalKind::FarmingReward{ farm_list, .. } => {
            vec![Default::default(); farm_list.len()]
        },
        ProposalKind::Poll{ options, .. } => {
            vec![Default::default(); options.len()]
        },
        ProposalKind::Common{ .. } => {
            vec![Default::default(); 3]
        }
    };

    let id = self.data().last_proposal_id;
```

```
        let proposal = Proposal{
            id,
            description,
            proposer: proposer.clone(),
            kind: kind.clone(),
            votes,
            ve_amount_at_last_action: self.data().cur_total_ve_lpt,
            incentive: HashMap::new(),
            start_at: to_nano(start_at),
            end_at: to_nano(start_at + duration_sec),
            participants: 0,
            status: None,
            is_nonsense: None
        };
        self.data_mut().proposals.insert(&id, &proposal.into());

        Event::ProposalCreate {
            proposer_id: &proposer,
            proposal_id: id,
            kind: &format!("{:?}", kind),
            start_at: to_nano(start_at),
            duration_sec
        }
        .emit();

        self.data_mut().last_proposal_id += 1;
        id
    }
```

Listing 2.2: contracts/ref-ve/src/actions_of_proposal.rs

Impact The contract is vulnerable to DoS attack. Malicious users can run out of storage by creating proposals with rather long description.

Suggestion I Limit the length of `Proposal.description` when creating proposals.

2.2 DeFi Security

2.2.1 User's Reward may be Lost

Status Fixed[1] in `Version 2`

Introduced by `Version 1`

Description When the `PromiseResult` is fail, there is no check on whether `sender_id` is registered. Function `callback_post_withdraw_reward` will panic if `sender_id` is not registered (line 78).

```
53    #[private]
54    pub fn callback_post_withdraw_reward(
55        &mut self,
56        token_id: AccountId,
57        sender_id: AccountId,
```

[1]This issue is fixed by recording the log and then manually distributing the rewards

```
        amount: U128,
    ) {
        require!(
            env::promise_results_count() == 1,
            E001_PROMISE_RESULT_COUNT_INVALID
        );
        let amount: Balance = amount.into();
        match env::promise_result(0) {
            PromiseResult::NotReady => unreachable!(),
            PromiseResult::Successful(_) => {
                Event::RewardWithdraw {
                    caller_id: &sender_id,
                    token_id: &token_id,
                    withdraw_amount: &U128(amount),
                    success: true,
                }
                .emit();
            }
            PromiseResult::Failed => {
                // This reverts the changes from withdraw function.
                let mut account = self.internal_unwrap_account(&sender_id);
                account.add_rewards(&HashMap::from([(token_id.clone(), amount)]));
                self.internal_set_account(&sender_id, account);

                Event::RewardWithdraw {
                    caller_id: &sender_id,
                    token_id: &token_id,
                    withdraw_amount: &U128(amount),
                    success: false,
                }
                .emit();
            }
        }
    }
```

Listing 2.3: contracts/ref-ve/src/actions_of_reward.rs

Impact If the `PromiseResult` is checked as failed and `sender_id` is unregistered, all rewards of this account(`sender_id`) will be lost.

Suggestion I It is suggested to check whether `sender_id` exists. If not, record the rewards of the `sender_id` in the `lostfound`.

2.2.2 Unreasonable Duration of Proposal

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is no limit check on the proposal's duration.

```
    #[payable]
    pub fn create_proposal(
        &mut self,
        kind: ProposalKind,
        description: String,
        start_at: u32,
        duration_sec: u32,
    ) -> u32 {
        let proposer = env::predecessor_account_id();
        require!(self.data().whitelisted_accounts.contains(&proposer) , E002_NOT_ALLOWED);

        self.internal_unwrap_account(&proposer);

        let config = self.internal_config();

        require!(start_at - nano_to_sec(env::block_timestamp()) >= config.
            min_proposal_start_vote_offset_sec, E402_INVALID_START_TIME);

        let votes: Vec<VoteInfo> = match &kind {
            ProposalKind::FarmingReward{ farm_list, .. } => {
                vec![Default::default(); farm_list.len()]
            },
            ProposalKind::Poll{ options, .. } => {
                vec![Default::default(); options.len()]
            },
            ProposalKind::Common{ .. } => {
                vec![Default::default(); 3]
            }
        };

        let id = self.data().last_proposal_id;
        let proposal = Proposal{
            id,
            description,
            proposer: proposer.clone(),
            kind: kind.clone(),
            votes,
            ve_amount_at_last_action: self.data().cur_total_ve_lpt,
            incentive: HashMap::new(),
            start_at: to_nano(start_at),
            end_at: to_nano(start_at + duration_sec),
            participants: 0,
            status: None,
            is_nonsense: None
        };
        self.data_mut().proposals.insert(&id, &proposal.into());

        Event::ProposalCreate {
            proposer_id: &proposer,
            proposal_id: id,
            kind: &format!("{:?}", kind),
            start_at: to_nano(start_at),
            duration_sec
        }
        .emit();

        self.data_mut().last_proposal_id += 1;
        id
    }
```

Listing 2.4: contracts/ref-ve/src/actions_of_proposal.rs

Impact The duration created for the voting period can be rather short (e.g., 1 block).

Suggestion I Limit the minimum duration seconds when creating proposals.

2.3 Additional Recommendation

2.3.1 Unused Function

Status Fixed in Version 2

Introduced by Version 1

Description Function `internal_set_proposal` is unused.

```
    pub fn internal_set_proposal(&mut self, proposal_id: u32, proposal: Proposal) {
        self.data_mut().proposals.insert(&proposal_id, &proposal.into());
    }
```

Listing 2.5: contracts/ref-ve/src/proposals.rs

Suggestion I Remove the unused functions.

2.3.2 Lack of Checking on the Locking Duration

Status Fixed in Version 2

Introduced by Version 1

Description There is no check on whether `min_locking_duration_sec` is smaller than `max_locking_duration_sec`. If owner or operators accidentally set `max_locking_duration_sec` to smaller than `min_locking_duration_sec`, then users cannot lock lpTokens.

```
    #[payable]
    pub fn modify_locking_policy(&mut self, min_duration: DurationSec, max_duration: DurationSec,
         max_ratio: u32) {
        assert_one_yocto();
        require!(self.is_owner_or_operators(), E002_NOT_ALLOWED);

        let mut config = self.data().config.get().unwrap();
        config.min_locking_duration_sec = min_duration;
        config.max_locking_duration_sec = max_duration;
        config.max_locking_multiplier = max_ratio;

        config.assert_valid();
        self.data_mut().config.set(&config);
    }
```

Listing 2.6: contracts/ref-ve/src/management.rs

```
impl Config {
    pub fn assert_valid(&self) {
        require!(
            self.max_locking_multiplier > MIN_LOCKING_REWARD_RATIO,
            E301_INVALID_RATIO
        );
    }
}
```

Listing 2.7: contracts/ref-ve/src/lib.rs

Suggestion I It is recommended to check whether `min_locking_duration_sec` is smaller than `max_locking_duration_sec` in function `assert_valid`.

2.3.3 Lack of assert_one_yocto()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `create_proposal` is a sensitive operation and function `assert_one_yocto()` should be added in function `create_proposal` for 2FA.

```
    #[payable]
    pub fn create_proposal(
        &mut self,
        kind: ProposalKind,
        description: String,
        start_at: u32,
        duration_sec: u32,
    ) -> u32 {
        let proposer = env::predecessor_account_id();
        require!(self.data().whitelisted_accounts.contains(&proposer) , E002_NOT_ALLOWED);

        self.internal_unwrap_account(&proposer);

        let config = self.internal_config();

        require!(start_at - nano_to_sec(env::block_timestamp()) >= config.
            min_proposal_start_vote_offset_sec, E402_INVALID_START_TIME);

        let votes: Vec<VoteInfo> = match &kind {
            ProposalKind::FarmingReward{ farm_list, .. } => {
                vec![Default::default(); farm_list.len()]
            },
            ProposalKind::Poll{ options, .. } => {
                vec![Default::default(); options.len()]
            },
            ProposalKind::Common{ .. } => {
                vec![Default::default(); 3]
            }
        };

        let id = self.data().last_proposal_id;
        let proposal = Proposal{
            id,
            description,
            proposer: proposer.clone(),
            kind: kind.clone(),
            votes,
            ve_amount_at_last_action: self.data().cur_total_ve_lpt,
            incentive: HashMap::new(),
            start_at: to_nano(start_at),
            end_at: to_nano(start_at + duration_sec),
            participants: 0,
            status: None,
            is_nonsense: None
        };
        self.data_mut().proposals.insert(&id, &proposal.into());

        Event::ProposalCreate {
            proposer_id: &proposer,
            proposal_id: id,
            kind: &format!("{:?}", kind),
            start_at: to_nano(start_at),
            duration_sec
        }
        .emit();

        self.data_mut().last_proposal_id += 1;
        id
    }
```

Listing 2.8: contracts/ref-ve/src/actions_of_proposal.rs

Suggestion I Add `assert_one_yocto()` in function `create_proposal`.

2.3.4 Lack of assert_one_yocto()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `action_proposal` is a sensitive operation and function `assert_one_yocto()` should be added in function `action_proposal` for 2FA.

```
    pub fn action_proposal(&mut self, proposal_id: u32, action: Action, memo: Option<String>) ->
        U128 {
        let voter = env::predecessor_account_id();

        let ve_lpt_amount = self.internal_account_vote(&voter, proposal_id, &action);

        self.internal_append_vote(proposal_id, &action, ve_lpt_amount);

        if let Some(memo) = memo {
            log!("Memo: {}", memo);
        }

        Event::ActionProposal {
            voter_id: &voter,
            proposal_id,
            action: &format!("{:?}", action)
        }
        .emit();

        ve_lpt_amount.into()
    }
```

Listing 2.9: contracts/ref-ve/src/actions_of_proposal.rs

Suggestion I Add `assert_one_yocto()` in function `action_proposal`.

2.3.5 Lack of Checking on the Gas Used by migrate

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is no check on whether `attached_gas` is enough for executing function `migrate`.

```
#[cfg(target_arch = "wasm32")]
mod upgrade {
    use near_sdk::Gas;
    use near_sys as sys;

    use super::*;

    /// Gas for calling migration call.
    pub const GAS_FOR_MIGRATE_CALL: Gas = Gas(5_000_000_000_000);

    /// Self upgrade and call migrate, optimizes gas by not loading into memory the code.
    /// Takes as input non serialized set of bytes of the code.
    #[no_mangle]
    pub fn upgrade() {
        env::setup_panic_hook();
        let contract: Contract = env::state_read().expect("ERR_CONTRACT_IS_NOT_INITIALIZED");
        contract.assert_owner();
        let current_id = env::current_account_id().as_bytes().to_vec();
        let method_name = "migrate".as_bytes().to_vec();
        unsafe {
            // Load input (wasm code) into register 0.
            sys::input(0);
            // Create batch action promise for the current contract ID
            let promise_id =
                sys::promise_batch_create(current_id.len() as _, current_id.as_ptr() as _);
            // 1st action in the Tx: "deploy contract" (code is taken from register 0)
            sys::promise_batch_action_deploy_contract(promise_id, u64::MAX as _, 0);
            // 2nd action in the Tx: call this_contract.migrate() with remaining gas
            let attached_gas = env::prepaid_gas() - env::used_gas() - GAS_FOR_MIGRATE_CALL;
            sys::promise_batch_action_function_call(
                promise_id,
                method_name.len() as _,
                method_name.as_ptr() as _,
                0 as _,
                0 as _,
                0 as _,
                attached_gas.0,
            );
        }
    }
}
```

Listing 2.10: contracts/ref-ve/src/owner.rs

Suggestion I Check whether `attached_gas` is larger than a specified value.

2.3.6 Potential Centralization Problem

Status Confirmed

Introduced by `Version 1`

Description This project has potential centralization problems. The project owner needs to ensure the security of the private key of `ContractData.owner_id` and use a multi-signature scheme to reduce the risk of single-point failure.

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

Feedback from the Project Yes, the owner is a DAO. That's why we import operator roles. It's a trade off result between security and effiecency

2.3.7 Potential Elastic Supply Token Problem

Status Confirmed

Introduced by `Version 1`

Description Elastic supply tokens (e.g., deflation tokens) could dynamically adjust the supply or user's balance. For example, if the token is a deflation token, there will be a difference between the transferred amount of tokens and the actual received amount of tokens.

This inconsistency can lead to security impacts for the operations based on the transferred amount of tokens instead of the actual received amount of tokens.

Suggestion I Do not append the elastic supply tokens into the whitelist.

Feedback from the Project Yes, we don't support elastic tokens for now.

2.4 Notes

2.4.1 Action::VoteNonsense is invalid

Status Confirmed

Introduced by `Version 1`

Description If users vote to `Action::VoteNonsense`, `E201_INVALID_VOTE` is triggered(line 83).

```
76  pub fn internal_append_vote(
77      &mut self,
78      proposal_id: u32,
79      action: &Action,
80      amount: Balance,
81  ) {
82      let mut proposal = self.internal_unwrap_proposal(proposal_id);
83      require!(action != &Action::VoteNonsense, E201_INVALID_VOTE);
84
85      // check proposal is inprogress
86      match proposal.status {
87          Some(ProposalStatus::InProgress) => {
88              // update proposal result
89              proposal.update_votes(
90                  action,
91                  amount,
92                  true
93              );
94              proposal.ve_amount_at_last_action = self.data().cur_total_ve_lpt;
95              proposal.votes[action.get_index()].participants += 1;
96              proposal.participants += 1;
97
98              self.data_mut()
99                  .proposals
100                 .insert(&proposal_id, &proposal.into());
101         },
102         _ => env::panic_str(E205_NOT_VOTABLE)
103     }
104 }
```

Listing 2.11: contracts/ref-ve/src/proposals_action.rs

Feedback from the Project At the beginning of the design, it was designed to support veToken holders to create proposals. A security deposit is required in case malicious proposals. The vote ratio of `Action::VoteNonsense` is used to determine whether to confiscate the security deposits. However, at this stage, only whitelisted users are allowed to create proposals. In this case, this option is temporarily unavailable.