Security Audit Report for NEP141 Token Vesting

Date: October 25, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2022.10.25
18:06:08 +08'00'

Contents

Report Manifest

Item	Description
Client	Octopus Network
Target	NEP141 Token Vesting

Version History

Version	Date	Description
1.0	October 25, 2022	First Release

About BlockSec The BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

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 the **NEP141 Token Vesting** contract [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 (`Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
NEP141 Token Vesting	`Version 1`	`3001151eba0b41f8baa6fbe8eac2cc6b39c0c5ec`
	`Version 2`	`2af0b12c483abda8db4d98b1f9b14ad3848dc106`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **nep141-token-vesting-contract/src** folder contract only. Specifically, the file covered in this audit include:

- beneficiary.rs
- constants.rs
- contract_viewers.rs
- events.rs
- external.rs
- fungible_token.rs
- interfaces.rs
- lib.rs
- owner.rs
- types.rs
- utils.rs
- vesting/
  - cliff.rs
  - linear.rs
  - mod.rs
  - traits.rs

[1] https://github.com/octopus-network/nep141-token-vesting

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.
- **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
- Permission management
- 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.

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.

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

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

Table 1.1: Vulnerability Severity Classification

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

- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **five** potential issues. We also have **nine** recommendations:

- High Risk: 2
- Medium Risk: 1
- Low Risk: 2
- Recommendations: 9

ID	Severity	Description	Category	Status
1	Low	Non-refundable Storage Fee	Software Security	Confirmed
2	High	Inconsistent Amount between Claimed Tokens and Transferred Tokens	DeFi Security	Fixed
3	High	Unexpected Removal of Vestings with Unclaimed Tokens	DeFi Security	Fixed
4	Low	Incomplete State Rollback	DeFi Security	Confirmed
5	Medium	Lack of Check while Removing Vestings	DeFi Security	Confirmed
6	-	Contract is not Upgradeable	Recommendation	Confirmed
7	-	Potential Centralization Problem	Recommendation	Confirmed
8	-	Potential Elastic Supply Token Problem	Recommendation	Confirmed
9	-	Two-Step Transfer of Privileged Account Ownership	Recommendation	Confirmed
10	-	Redundant Code	Recommendation	Fixed
11	-	Lack of Check When Changing the Beneficiary	Recommendation	Fixed
12	-	Lack of Check When Creating Vestings	Recommendation	Acknowledged
13	-	Missed Limitation on the Number of Cliff Checkpoints	Recommendation	Confirmed
14	-	Improper Checks on the End of Vestings	Recommendation	Fixed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Non-refundable Storage Fee

Severity Low

Status Confirmed

Introduced by Version 1

Description When vestings are removed from the contract, the corresponding storage fee deposited before will not be refunded.

```
#[near_bindgen]
impl TokenVestingContract {
    #[private]
    pub fn claim_callback(
        &mut self,
        vesting_id: VestingId,
        amount: Option<U128>,
        #[callback_unwrap] ft_balance: U128,
        #[callback_unwrap] storage_balance: Option<StorageBalance>,
    ) -> U128 {
        assert!(
            storage_balance.is_some(),
            "Failed to claim because the beneficiary hasn’t registered in vesting token contract."
        );

        let mut vesting = self
            .internal_get_vesting(&vesting_id)
            .expect(format!("Failed to claim, no such vesting id: #{}", vesting_id.0).as_str());
        let beneficiary = vesting.get_beneficiary();
        let claimable_amount = vesting.claim(amount.map(|e| e.0));

        assert!(
            ft_balance.0 >= claimable_amount,
            "Failed to claim because the contract balance is not enough."
        );

        if vesting.is_release_finish() {
            self.internal_remove_vesting(&vesting_id);
            VestingEvent::FinishVesting {
                vesting_id: &vesting_id,
            }
            .emit();
        } else {
            self.internal_save_vesting(&vesting);
        }

        VestingEvent::UpdateVesting { vesting: &vesting }.emit();
        let transfer_id = self.internal_assign_id();

        UserAction::Claim {
            transfer_id: &transfer_id,
            vesting_id: &vesting_id,
            beneficiary: &beneficiary,
            token_id: &self.token_id,
            amount: &U128(claimable_amount),
        }
        .emit();

        self.internal_send_tokens(
            &beneficiary,
            &self.token_id.clone(),
            claimable_amount,
            Some(transfer_id),
        );
        U128(claimable_amount)
    }
```

Listing 2.1: nep141-token-vesting-contract/src/beneficiary.rs

```
    #[private]
    pub fn claim_all_callback(
        &mut self,
        beneficiary: AccountId,
        #[callback_unwrap] ft_balance: U128,
        #[callback_unwrap] storage_balance: Option<StorageBalance>,
    ) -> U128 {
        assert!(
            storage_balance.is_some(),
            "Failed to claim because the beneficiary hasn’t registered in vesting token contract."
        );

        let mut amount: u128 = 0;
        let vestings = self
            .vestings
            .values()
            .filter(|e| e.get_beneficiary().eq(&beneficiary))
            .collect_vec();

        let mut claimed_vesting_ids: Vec<VestingId> = vec![];
        for mut vesting in vestings {
            let vesting_id = vesting.get_vesting_id();
            let claimable_amount = vesting.claim(Option::None);

            if claimable_amount == 0 {
                continue;
            }

            if vesting.is_release_finish() {
                self.internal_remove_vesting(&vesting_id);
                VestingEvent::FinishVesting {
                    vesting_id: &vesting_id,
                }
                .emit();
            } else {
                self.internal_save_vesting(&vesting)
            }

            VestingEvent::UpdateVesting { vesting: &vesting }.emit();

            amount += claimable_amount;
            claimed_vesting_ids.push(vesting_id);
        }

        if amount > 0 {
            assert!(
                ft_balance.0 >= amount,
                "Failed to claim because the contract balance is not enough."
            );

            let transfer_id = self.internal_assign_id();

            UserAction::ClaimAll {
```

```
            transfer_id: &transfer_id,
            vesting_ids: &claimed_vesting_ids,
            beneficiary: &beneficiary,
            token_id: &self.token_id.clone(),
            amount: &U128(amount),
        }
        .emit();

        self.internal_send_tokens(
            &beneficiary,
            &self.token_id.clone(),
            amount,
            Some(transfer_id),
        );
    }
    U128(amount)
}
```

Listing 2.2: nep141-token-vesting-contract/src/beneficiary.rs

```
fn terminate_vesting(&mut self, vesting_id: VestingId) {
    self.assert_owner();

    self.vestings.remove(&vesting_id);

    UserAction::TerminateVesting {
        vesting_id: &vesting_id,
    }
    .emit();
}
```

Listing 2.3: nep141-token-vesting-contract/src/owner.rs

In addition, in function `change_beneficiary()`, if the length of the new beneficiary's `AccountId` is shorter than the previous one, the storage fee cannot be refunded either.

```
#[payable]
fn change_beneficiary(&mut self, vesting_id: VestingId, new_beneficiary: AccountId) {
    let prev_storage = env::storage_usage();

    let mut vesting = self
        .internal_get_vesting(&vesting_id)
        .expect("No such vesting.");
    assert!(
        env::predecessor_account_id().eq(&vesting.get_beneficiary())
            || env::predecessor_account_id().eq(&self.owner),
        "Only owner and vesting beneficiary can set a new beneficiary."
    );
    let old_beneficiary = vesting.get_beneficiary();

    vesting.set_beneficiary(new_beneficiary);

    self.internal_save_vesting(&vesting);

```

```
        self.internal_check_storage(prev_storage);

        VestingEvent::UpdateVesting {
            vesting: &self
                .internal_get_vesting(&vesting_id)
                .expect(format!("Failed to get vesting by id: {}.", &vesting_id.0).as_str()),
        }
        .emit();

        UserAction::ChangeBeneficiary {
            vesting_id: &vesting_id,
            old_beneficiary: &old_beneficiary,
            new_beneficiary: &vesting.get_beneficiary(),
        }
        .emit();
    }
```

Listing 2.4: nep141-token-vesting-contract/src/beneficiary.rs

Impact The staked storage fee will be locked after the storage is released.

Suggestion If the contract storage is released, refund the storage fee.

2.2 DeFi Security

2.2.1 Inconsistent Amount between Claimed Tokens and Transferred Tokens

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `claim_callback()`, the amount of tokens transferred to the beneficiary (line 131) is returned from function `vesting.claim()`(line 99).

```
80    #[near_bindgen]
81    impl TokenVestingContract {
82        #[private]
83        pub fn claim_callback(
84            &mut self,
85            vesting_id: VestingId,
86            amount: Option<U128>,
87            #[callback_unwrap] ft_balance: U128,
88            #[callback_unwrap] storage_balance: Option<StorageBalance>,
89        ) -> U128 {
90            assert!(
91                storage_balance.is_some(),
92                "Failed to claim because the beneficiary hasn't registered in vesting token
                      contract."
93            );
94
95            let mut vesting = self
96                .internal_get_vesting(&vesting_id)
```

```
                .expect(format!("Failed to claim, no such vesting id: #{}", vesting_id.0).as_str())
                    ;
            let beneficiary = vesting.get_beneficiary();
            let claimable_amount = vesting.claim(amount.map(|e| e.0));

            assert!(
                ft_balance.0 >= claimable_amount,
                "Failed to claim because the contract balance is not enough."
            );

            if vesting.is_release_finish() {
                self.internal_remove_vesting(&vesting_id);
                VestingEvent::FinishVesting {
                    vesting_id: &vesting_id,
                }
                .emit();
            } else {
                self.internal_save_vesting(&vesting);
            }

            VestingEvent::UpdateVesting { vesting: &vesting }.emit();
            let transfer_id = self.internal_assign_id();

            UserAction::Claim {
                transfer_id: &transfer_id,
                vesting_id: &vesting_id,
                beneficiary: &beneficiary,
                token_id: &self.token_id,
                amount: &U128(claimable_amount),
            }
            .emit();

            self.internal_send_tokens(
                &beneficiary,
                &self.token_id.clone(),
                claimable_amount,
                Some(transfer_id),
            );
            U128(claimable_amount)
        }
```

Listing 2.5: nep141-token-vesting-contract/src/beneficiary.rs

Meanwhile, for each claim, the beneficiary is allowed to request any amount of vesting tokens no greater than the `claimable_amount`, and the amount will be accumulated to the claimed tokens for this `vesting` (lines 86-88).

```
72    impl<T: VestingAmount + VestingTokenInfoTrait + Frozen> Claimable for T {
73        fn claim(&mut self, amount: Option<Balance>) -> Balance {
74            assert!(
75                !self.is_frozen(),
76                "Failed to claim because this vesting is frozen."
77            );
78            let claimable_amount = self.get_claimable_amount();
79          if amount.is_some() {
80              assert!(
81                  amount.unwrap() <= claimable_amount,
82                  "claimable amount is less than claim amount."
83              );
84          }
85
86          self.set_claimed_token_amount(
87              self.get_vesting_token_info().claimed_token_amount + amount.unwrap_or(
                    claimable_amount),
88          );
89          claimable_amount
90      }
91  }
```

Listing 2.6: nep141-token-vesting-contract/src/vesting/mod.rs

However, the amount returned from the function `vesting.claim()` is always `claimable_amount` (line 89) while the recorded claimed token is `amount`. Note that the amount can be rather smaller than `claimable_amount`. In this case, the contract may send more vesting tokens to the beneficiary than expected.

Impact The vesting tokens of this contract can be drained.

Suggestion If the parameter `amount` is `Some` in function `vesting.claim()`, return the value of the amount instead of the `claimable_amount`.

2.2.2 Unexpected Removal of Vestings with Unclaimed Tokens

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `claim()` allows the beneficiary to claim a portion of the claimable tokens from a specific `vesting`.

```
fn claim(&mut self, vesting_id: VestingId, amount: Option<U128>) -> PromiseOrValue<U128> {
    let vesting = self
        .internal_get_vesting(&vesting_id)
        .expect("No such vesting,id: #{}.");

    PromiseOrValue::Promise(
        ext_ft_core::ext(self.token_id.clone())
            .ft_balance_of(current_account_id())
            .and(
                ext_storage_management::ext(self.token_id.clone())
                    .storage_balance_of(vesting.get_beneficiary()),
            )
            .then(Self::ext(env::current_account_id()).claim_callback(vesting_id, amount)),
    )
}
```

Listing 2.7: nep141-token-vesting-contract/src/beneficiary.rs

However, if the release of the `vesting` is checked as finished (line 106), the contract will remove the `vesting` directly without checking whether there exist unclaimed tokens in the `vesting`.

```
82      #[private]
83      pub fn claim_callback(
84          &mut self,
85          vesting_id: VestingId,
86          amount: Option<U128>,
87          #[callback_unwrap] ft_balance: U128,
88          #[callback_unwrap] storage_balance: Option<StorageBalance>,
89      ) -> U128 {
90          assert!(
91              storage_balance.is_some(),
92              "Failed to claim because the beneficiary hasn't registered in vesting token contract."
93          );
94
95          let mut vesting = self
96              .internal_get_vesting(&vesting_id)
97              .expect(format!("Failed to claim, no such vesting id: #{}", vesting_id.0).as_str());
98          let beneficiary = vesting.get_beneficiary();
99          let claimable_amount = vesting.claim(amount.map(|e| e.0));
100
101         assert!(
102             ft_balance.0 >= claimable_amount,
103             "Failed to claim because the contract balance is not enough."
104         );
105
106         if vesting.is_release_finish() {
107             self.internal_remove_vesting(&vesting_id);
108             VestingEvent::FinishVesting {
109                 vesting_id: &vesting_id,
110             }
111             .emit();
112         } else {
113             self.internal_save_vesting(&vesting);
114         }
115
116         VestingEvent::UpdateVesting { vesting: &vesting }.emit();
117         let transfer_id = self.internal_assign_id();
118
119         UserAction::Claim {
120             transfer_id: &transfer_id,
121             vesting_id: &vesting_id,
122             beneficiary: &beneficiary,
123             token_id: &self.token_id,
124             amount: &U128(claimable_amount),
125         }
126         .emit();
127
128         self.internal_send_tokens(
129             &beneficiary,
130             &self.token_id.clone(),
131             claimable_amount,
        Some(transfer_id),
    );
    U128(claimable_amount)
}
```

Listing 2.8: nep141-token-vesting-contract/src/beneficiary.rs

Impact The beneficiary's funds will be lost.

Suggestion Ensure all vesting tokens are claimed by users before removing the `vesting`. Therefore, another function named `is_vesting_finish()` is suggested to be used.

```
pub trait Finish: VestingTokenInfoTrait {
    fn is_release_finish(&self) -> bool;
    fn is_vesting_finish(&self) -> bool {
        self.get_vesting_token_info().total_vesting_amount
            == self.get_vesting_token_info().claimed_token_amount
    }
}
```

Listing 2.9: nep141-token-vesting-contract/src/vesting/traits.rs

2.2.3 Incomplete State Rollback

Severity Low

Status Confirmed

Introduced by `Version 1`

Description According to the current implementation of the callback function `ft_transfer_resolved()`, the updated `claimed_token_amount` for a specific `vesting` cannot be recovered if the `PromiseResult` of the cross-contract invocation `ft_transfer()` is checked as failed.

```
#[near_bindgen]
impl TokenVestingContract {
    #[private]
    pub fn claim_callback(
        &mut self,
        vesting_id: VestingId,
        amount: Option<U128>,
        #[callback_unwrap] ft_balance: U128,
        #[callback_unwrap] storage_balance: Option<StorageBalance>,
    ) -> U128 {
        assert!(
            storage_balance.is_some(),
            "Failed to claim because the beneficiary hasn't registered in vesting token
                 contract."
        );

        let mut vesting = self
            .internal_get_vesting(&vesting_id)
            .expect(format!("Failed to claim, no such vesting id: #{}", vesting_id.0).as_str())
                 ;
        let beneficiary = vesting.get_beneficiary();
        let claimable_amount = vesting.claim(amount.map(|e| e.0));

        assert!(
            ft_balance.0 >= claimable_amount,
            "Failed to claim because the contract balance is not enough."
        );

        if vesting.is_release_finish() {
            self.internal_remove_vesting(&vesting_id);
            VestingEvent::FinishVesting {
                vesting_id: &vesting_id,
            }
            .emit();
        } else {
            self.internal_save_vesting(&vesting);
        }

        VestingEvent::UpdateVesting { vesting: &vesting }.emit();
        let transfer_id = self.internal_assign_id();

        UserAction::Claim {
            transfer_id: &transfer_id,
            vesting_id: &vesting_id,
            beneficiary: &beneficiary,
            token_id: &self.token_id,
            amount: &U128(claimable_amount),
        }
        .emit();

        self.internal_send_tokens(
            &beneficiary,
            &self.token_id.clone(),
            claimable_amount,
            Some(transfer_id),
        );
        U128(claimable_amount)
    }
```

Listing 2.10: nep141-token-vesting-contract/src/beneficiary.rs

```
pub(crate) fn internal_send_tokens(
    &mut self,
    receiver_id: &AccountId,
    token_id: &AccountId,
    amount: Balance,
    transfer_id: Option<TransferId>,
) {
    assert!(amount > 0, "Failed to send tokens because amount is 0.");
    ext_ft_core::ext(token_id.clone())
        .with_attached_deposit(ONE_YOCTO)
        .with_static_gas(Gas::ONE_TERA.mul(T_GAS_FOR_FT_TRANSFER))
        .ft_transfer(receiver_id.clone(), U128(amount), None)
        .then(
            Self::ext(env::current_account_id())
                .with_static_gas(Gas::ONE_TERA.mul(T_GAS_FOR_RESOLVE_TRANSFER))
                .ft_transfer_resolved(
                    token_id.clone(),
                    receiver_id.clone(),
                    U128(amount),
                    transfer_id,
                ),
        );
}
```

Listing 2.11: nep141-token-vesting-contract/src/fungible_token.rs

```
#[private]
pub fn ft_transfer_resolved(
    &mut self,
    token_id: AccountId,
    receiver_id: AccountId,
    amount: U128,
    transfer_id: Option<TransferId>,
) {
    assert_eq!(
        env::promise_results_count(),
        1,
        "Expect 1 promise result for ft_transfer_resolved."
    );
    log!(
        "ft_transfer_resolved, token_id: {}, receiver_id: {}, amount: {}",
        token_id,
        receiver_id,
        amount.0
    );
    match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Successful(_) => {
            if transfer_id.is_some() {
                ActionStatus::FtTransferResult {
                    transfer_id: &transfer_id.unwrap(),
                    is_success: &true,
                }
                .emit()
            }
        }
        PromiseResult::Failed => {
            if transfer_id.is_some() {
                ActionStatus::FtTransferResult {
                    transfer_id: &transfer_id.unwrap(),
                    is_success: &false,
                }
                .emit()
            }

            UserAction::Legacy {
                account_id: &receiver_id,
                token_id: &self.token_id,
                amount: &amount,
            }
            .emit();
        }
    }
}
```

Listing 2.12: nep141-token-vesting-contract/src/fungible_token.rs

Impact User's fund will be lost due to the incomplete state rollback.

Suggestion Recover the `claimed_token_amount` of `vesting` if the token transfer is failed.

2.2.4 Lack of Check while Removing Vestings

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Vestings can be removed by the owner directly even if there exists `claimable tokens` that are not claimed by the `beneficiary` yet.

```
fn terminate_vesting(&mut self, vesting_id: VestingId) {
    self.assert_owner();

    self.vestings.remove(&vesting_id);

    UserAction::TerminateVesting {
        vesting_id: &vesting_id,
    }
    .emit();
}
```

Listing 2.13: nep141-token-vesting-contract/src/owner.rs

Impact It is unfair for the `beneficiary`, who doesn't claim all the `claimable tokens` in time before the termination.

Suggestion Transfer the `claimable tokens` to the `beneficiary` when terminating the `vesting` with function `terminate_vesting()`.

2.3 Additional Recommendation

2.3.1 Contract is not Upgradeable

Status Confirmed

Introduced by `Version 1`

Description There is no contract upgrade function within the audit scope. Thus, the contract can not be upgraded for further enhancements or patches.

In addition, the person with the full access key to this contract AccountId may be able to upgrade the contract directly and transfer the vesting tokens out, which may lead to a potential centralization problem. Suggestion Implement a privileged function for the contract upgrade.

Suggestion I Implement a privileged function for the contract upgrade.

Feedback from the Project We have considered the case. We prefer to keep current implementation.

2.3.2 Potential Centralization Problem

Status Confirmed

Introduced by Version 1

Description This project has potential centralization problems. The TokenVestingContract.owner has the privilege to configure a number of system parameters (e.g., changing the beneficiary of a specific vesting) and freezing or terminating vesting.

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 This is intentional. This contract is not for trustless case. The rights of owner is by design.

2.3.3 Potential Elastic Supply Token Problem

Status Confirmed

Introduced by Version 1

Description Elastic supply tokens could dynamically adjust their price, supply, user's balance, etc. For example, inflation tokens, deflation tokens, rebasing tokens, and so forth.

In the current contract implementation, elastic supply tokens are not supported. If the token is a deflation token, there will be a difference between the recorded amount of transferred tokens to the beneficiary (as a parameter of function ft_transfer()) and the actual number of transferred tokens (the token smart contract itself). That's because the token smart contract will burn a small number of tokens.

Suggestion I Do not set TokenVestingContract.token_id to an elastic supply token.

Feedback from the Project Noted. We'll not use elastic supply token in this contract.

2.3.4 Two-Step Transfer of Privileged Account Ownership

Status Confirmed

Introduced by Version 1

Description The contract uses set_owner() to configure the privileged account, which can conduct many sensitive operations (e.g., create vestings). In this case, when an incorrect new owner is provided, the contract is at risk of attack and the privileged function cannot be invoked.

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

Listing 2.14: nep141-token-vesting-contract/src/owner.rs

Suggestion I Implement a two-step approach for the `owner` update: `propose_owner()` and `accept_owner()`.

Feedback from the Project We'll take the risk. No need to change.

2.3.5 Redundant Code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `transfer_id` won't be `None` in function `ft_transfer_resolved()`. This is because callers of `ft_transfer_resolved()` (`claim_callback()` and `claim_all_callback()`) will always set this `parameter`.

```
    #[private]
    pub fn ft_transfer_resolved(
        &mut self,
        token_id: AccountId,
        receiver_id: AccountId,
        amount: U128,
        transfer_id: Option<TransferId>,
    ) {
        assert_eq!(
            env::promise_results_count(),
            1,
            "Expect 1 promise result for ft_transfer_resolved."
        );
        log!(
            "ft_transfer_resolved, token_id: {}, receiver_id: {}, amount: {}",
            token_id,
            receiver_id,
            amount.0
        );
        match env::promise_result(0) {
            PromiseResult::NotReady => unreachable!(),
            PromiseResult::Successful(_) => {
                if transfer_id.is_some() {
                    ActionStatus::FtTransferResult {
                        transfer_id: &transfer_id.unwrap(),
                        is_success: &true,
                    }
                    .emit()
                }
            }
            PromiseResult::Failed => {
                if transfer_id.is_some() {
                    ActionStatus::FtTransferResult {
                        transfer_id: &transfer_id.unwrap(),
                        is_success: &false,
                    }
                    .emit()
                }

                UserAction::Legacy {
                    account_id: &receiver_id,
                    token_id: &self.token_id,
                    amount: &amount,
                }
                .emit();
            }
        }
    }
```

Listing 2.15: nep141-token-vesting-contract/src/fungible_token.rs

```
#[near_bindgen]
impl TokenVestingContract {
    #[private]
    pub fn claim_callback(
        &mut self,
        vesting_id: VestingId,
        amount: Option<U128>,
        #[callback_unwrap] ft_balance: U128,
        #[callback_unwrap] storage_balance: Option<StorageBalance>,
    ) -> U128 {
        assert!(
            storage_balance.is_some(),
            "Failed to claim because the beneficiary hasn’t registered in vesting token contract."
        );

        let mut vesting = self
            .internal_get_vesting(&vesting_id)
            .expect(format!("Failed to claim, no such vesting id: #{}", vesting_id.0).as_str());
        let beneficiary = vesting.get_beneficiary();
        let claimable_amount = vesting.claim(amount.map(|e| e.0));

        assert!(
            ft_balance.0 >= claimable_amount,
            "Failed to claim because the contract balance is not enough."
        );

        if vesting.is_release_finish() {
            self.internal_remove_vesting(&vesting_id);
            VestingEvent::FinishVesting {
                vesting_id: &vesting_id,
            }
            .emit();
        } else {
            self.internal_save_vesting(&vesting);
        }

        VestingEvent::UpdateVesting { vesting: &vesting }.emit();
        let transfer_id = self.internal_assign_id();

        UserAction::Claim {
            transfer_id: &transfer_id,
            vesting_id: &vesting_id,
            beneficiary: &beneficiary,
            token_id: &self.token_id,
            amount: &U128(claimable_amount),
        }
        .emit();

        self.internal_send_tokens(
            &beneficiary,
            &self.token_id.clone(),
            claimable_amount,
            Some(transfer_id),
        );
        U128(claimable_amount)
    }
```

Listing 2.16: nep141-token-vesting-contract/src/beneficiary.rs

Suggestion I Remove the redundant statement in line 58 and line 67 in function `ft_transfer_resolved()`.

2.3.6 Lack of Check When Changing the Beneficiary

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `change_beneficiary()`, there is no check on whether the `new_beneficiary` is the same as the previous one.

```
#[payable]
fn change_beneficiary(&mut self, vesting_id: VestingId, new_beneficiary: AccountId) {
    let prev_storage = env::storage_usage();

    let mut vesting = self
        .internal_get_vesting(&vesting_id)
        .expect("No such vesting.");
    assert!(
        env::predecessor_account_id().eq(&vesting.get_beneficiary())
            || env::predecessor_account_id().eq(&self.owner),
        "Only owner and vesting beneficiary can set a new beneficiary."
    );
    let old_beneficiary = vesting.get_beneficiary();

    vesting.set_beneficiary(new_beneficiary);

    self.internal_save_vesting(&vesting);

    self.internal_check_storage(prev_storage);

```

```
        VestingEvent::UpdateVesting {
            vesting: &self
                .internal_get_vesting(&vesting_id)
                .expect(format!("Failed to get vesting by id: {}.", &vesting_id.0).as_str()),
        }
        .emit();

        UserAction::ChangeBeneficiary {
            vesting_id: &vesting_id,
            old_beneficiary: &old_beneficiary,
            new_beneficiary: &vesting.get_beneficiary(),
        }
        .emit();
    }
```

Listing 2.17: nep141-token-vesting-contract/src/beneficiary.rs

Suggestion I Add Checks in function `change_beneficiary()` to ensure that the `new_beneficiary` and the previous beneficiary are not the same.

2.3.7 Lack of Check When Creating Vestings

Status Acknowledged

Introduced by `Version 1`

Description In the function `Vesting::new()`, when creating a `LinearVesting`, there is no check for the parameter `start_time` to ensure it is later than the current timestamp.

What's more, when the `CliffVesting` is created, all cliff `checkpoints` should also be checked to ensure the release time for each checkpoint is later than the current timestamp.

```
    pub fn new(id: VestingId, param: VestingCreateParam) -> Self {
        match param {
            VestingCreateParam::LinearVesting {
                beneficiary,
                start_time,
                end_time,
                total_vesting_amount,
            } => {
                assert!(start_time<end_time, "End time should be less than start time when creating NaturalTimeLinearVesting.");

                Vesting::NaturalTimeLinearVesting(NaturalTimeLinearVesting {
                    id,
                    beneficiary,
                    start_time,
                    end_time,
                    vesting_token_info: VestingTokenInfo {
                        claimed_token_amount: 0,
                        total_vesting_amount,
                    },
                    is_frozen: false,
                    create_time: get_block_second_time(),
                })
            }
            VestingCreateParam::CliffVesting {
                beneficiary,
                time_cliff_list,
            } => {
                let total_amount = time_cliff_list
                    .iter()
                    .map(|e| e.amount)
                    .reduce(|acc, item| {
                        acc.checked_add(item)
                            .expect("accumulation of cliff amount is overflow.")
                    })
                    .unwrap_or(0);
                Vesting::TimeCliffVesting(TimeCliffVesting {
                    id,
                    beneficiary,
                    time_cliff_list,
                    vesting_token_info: VestingTokenInfo {
                        claimed_token_amount: 0,
                        total_vesting_amount: total_amount,
                    },
                    is_frozen: false,
                    create_time: get_block_second_time(),
                })
            }
        }
    }
```

Listing 2.18: nep141-token-vesting-contract/src/vesting/mod.rs

Suggestion I Add sanity checks in function `Vesting::new()`.

Feedback from the Project We need to allow to vest for a passed time range. The checking should not be added.

2.3.8 Missed Limitation on the Number of Cliff Checkpoint

Status Confirmed

Introduced by `Version 1`

Description When creating a `CliffVesting` in function `Vesting::new()`, the number of checkpoints is unlimited. In this case, functions like `TimeCliffVesting.get_unreleased_amount()` and `TimeCliffVesting.is_release_finish()` may run out of gas for iterating through too many checkpoints.

```
pub fn new(id: VestingId, param: VestingCreateParam) -> Self {
    match param {
        VestingCreateParam::LinearVesting {
            beneficiary,
            start_time,
            end_time,
            total_vesting_amount,
        } => {
                assert!(start_time<end_time, "End time should be less than start time when creating NaturalTimeLinearVesting.");

                Vesting::NaturalTimeLinearVesting(NaturalTimeLinearVesting {
                    id,
                    beneficiary,
                    start_time,
                    end_time,
                    vesting_token_info: VestingTokenInfo {
                        claimed_token_amount: 0,
                        total_vesting_amount,
                    },
                    is_frozen: false,
                    create_time: get_block_second_time(),
                })
            }
            VestingCreateParam::CliffVesting {
                beneficiary,
                time_cliff_list,
            } => {
                let total_amount = time_cliff_list
                    .iter()
                    .map(|e| e.amount)
                    .reduce(|acc, item| {
                        acc.checked_add(item)
                            .expect("accumulation of cliff amount is overflow.")
                    })
                    .unwrap_or(0);
                Vesting::TimeCliffVesting(TimeCliffVesting {
                    id,
                    beneficiary,
                    time_cliff_list,
                    vesting_token_info: VestingTokenInfo {
                        claimed_token_amount: 0,
                        total_vesting_amount: total_amount,
                    },
                    is_frozen: false,
                    create_time: get_block_second_time(),
                })
            }
        }
    }
```

Listing 2.19: nep141-token-vesting-contract/src/vesting/mod.rs

```
impl Finish for TimeCliffVesting {
    fn is_release_finish(&self) -> bool {
        let max_time = self
            .time_cliff_list
            .iter()
            .map(|e| e.time)
            .max()
            .unwrap_or(0);
        return max_time < get_block_second_time();
    }
}
```

Listing 2.20: nep141-token-vesting-contract/src/vesting/cliff.rs

```
impl VestingAmount for TimeCliffVesting {
    fn get_unreleased_amount(&self) -> Balance {
        self.time_cliff_list
            .iter()
            .map(|e| {
                if e.time > get_block_second_time() {
                    e.amount
                } else {
                    0
                }
            })
            .sum()
    }
}
```

Listing 2.21: nep141-token-vesting-contract/src/vesting/cliff.rs

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

Feedback from the Project We prefer to keep current implementation.

2.3.9 Improper Checks on the End of Vestings

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `is_release_finish()` for `CliffVesting`, vesting will not be considered as ended if the current `block_second_time` is equal to `max_time`.

However, all the vesting tokens are released at the time of `max_time` according to the implementation of function `get_unreleased_amount()`. Thus, the finish of the release should be `max_time` instead of the time later than it.

```
impl Finish for TimeCliffVesting {
    fn is_release_finish(&self) -> bool {
        let max_time = self
            .time_cliff_list
            .iter()
            .map(|e| e.time)
            .max()
            .unwrap_or(0);
        return max_time < get_block_second_time();
    }
}
```

Listing 2.22: nep141-token-vesting-contract/src/vesting/cliff.rs

```
impl VestingAmount for TimeCliffVesting {
    fn get_unreleased_amount(&self) -> Balance {
        self.time_cliff_list
            .iter()
            .map(|e| {
                if e.time > get_block_second_time() {
                    e.amount
                } else {
                    0
                }
            })
            .sum()
    }
}
```

Listing 2.23: nep141-token-vesting-contract/src/vesting/cliff.rs

The same problem also exists in the function `is_release_finish()` for LinearVesting.

```
impl Finish for NaturalTimeLinearVesting {
    fn is_release_finish(&self) -> bool {
        self.end_time < get_block_second_time()
    }
}
```

Listing 2.24: nep141-token-vesting-contract/src/vesting/linear.rs

Suggestion I It is suggested to change the comparison operators mentioned above from "<" to "<=".