Security Audit Report for XRef Token Contract

Date: October 12, 2022

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2022.10.24 11:16:45 +08'00'

Contents

Report Manifest

Item	Description
Client	Ref-Finance
Target	XRef Token Contract

Version History

Version	Date	Description
1.0	October 12, 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 the **XRef Token** 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
XRef Token Contract	`Version 1`	`1f92f64c3ee773a42649d400c91b1decf322f953`

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

- lib.rs
- owner.rs
- storage_impl.rs
- utils.rs
- views.rs
- xref.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 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.

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

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.

Table 1.1: Vulnerability Severity Classification

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

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.

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

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

Chapter 2 Findings

In total, we find **three** potential issues. We also have **four** recommendations and **two** notes as follows:

- High Risk: 0
- Medium Risk: 0
- Low Risk: 3
- Recommendations: 4
- Notes: 2

ID	Severity	Description	Category	Status
1	Low	Improper Transfer Failure Handling during Unstaking	DeFi Security	Confirmed
2	Low	Unrecoverable Tokens for the Unregistered Account	DeFi Security	Confirmed
3	Low	Optional Reward Distribution before Modifying Reward Rate	DeFi Security	Confirmed
4	-	Potential Elastic Supply Token Problem	Recommendation	Confirmed
5	-	Lack of Check on the Success of Upgrade	Recommendation	Confirmed
6	-	Lack of Check on the Gas Used by migrate()	Recommendation	Confirmed
7	-	Improper Log Emission	Recommendation	Confirmed
8	-	Assumption on the Secure Implementation of Dependencies	Notes	Confirmed
9	-	Assumption on the Secure Management of the DAO	Notes	Confirmed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Improper Transfer Failure Handling during Unstaking

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Function `callback_post_unstake()` will recover the amount of locked token (i.e., `REF`) when the cross-contract invocation `ft_transfer()` executed in block N is failed. However, function `callback_post_unstake()` is executed in block N+1. The exchange ratio between `xREF` token and `REF` token may be different between block N+1 and block N. Therefore, it's unreasonable to mint the outdated shares (i.e., `xREF`) back to the user in line 120.

```
100     #[private]
101     pub fn callback_post_unstake(
102         &mut self,
103         sender_id: AccountId,
104         amount: U128,
105         share: U128,
106     ) {
107         assert_eq!(
108             env::promise_results_count(),
109             1,
            "Err: expected 1 promise result from unstake"
        );
        match env::promise_result(0) {
            PromiseResult::NotReady => unreachable!(),
            PromiseResult::Successful(_) => {}
            PromiseResult::Failed => {
                // This reverts the changes from unstake function.
                // If account doesn't exit, the unlock token stay in contract.
                if self.ft.accounts.contains_key(&sender_id) {
                    self.locked_token_amount += amount.0;
                    self.ft.internal_deposit(&sender_id, share.0);
                    env::log(
                        format!(
                            "Account {} unstake failed and reverted.",
                            sender_id
                        )
                        .as_bytes(),
                    );
                } else {
                    env::log(
                        format!(
                            "Account {} has unregisterd. unlocking token goes to contract.",
                            sender_id
                        )
                        .as_bytes(),
                    );
                }
            }
        };
    }
```

Listing 2.1: xref-token/src/xref.rs

Impact Users may get incorrect shares in function `callback_post_unstake()`.

Suggestion Re-calculate the shares (i.e., `xREF`) in function `callback_post_unstake()`.

Feedback from the Project From the user's perspective, he just suffered a failed unstake, what he wants is to have his assets integrity. If we recalculate `xREF` amount based on market at callback execution moment, users would be quite possible to get different amounts of `xREF`. If the amount is less, users won't care about the inner logic, and must issue complaints which turns out to be our big trouble. So, it's necessary to keep the amount unchanged, which won't cause any assets loss to all `xREF` users.

2.1.2 Unrecoverable Tokens for the Unregistered Account

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In function `callback_post_unstake()`, if the `PromiseResult` is checked as `Failed` and the sender's account is unregistered, the contract will not recover the state changed in function `unstake()`.

Instead, a log without specifying the unstaked amount is emitted in lines 129-135, which cannot help to recover the users' assets.

```
100     #[private]
101     pub fn callback_post_unstake(
102         &mut self,
103         sender_id: AccountId,
104         amount: U128,
105         share: U128,
106     ) {
107         assert_eq!(
108             env::promise_results_count(),
109             1,
110             "Err: expected 1 promise result from unstake"
111         );
112         match env::promise_result(0) {
113             PromiseResult::NotReady => unreachable!(),
114             PromiseResult::Successful(_) => {}
115             PromiseResult::Failed => {
116                 // This reverts the changes from unstake function.
117                 // If account doesn't exit, the unlock token stay in contract.
118                 if self.ft.accounts.contains_key(&sender_id) {
119                     self.locked_token_amount += amount.0;
120                     self.ft.internal_deposit(&sender_id, share.0);
121                     env::log(
122                         format!(
123                             "Account {} unstake failed and reverted.",
124                             sender_id
125                         )
126                         .as_bytes(),
127                     );
128                 } else {
129                     env::log(
130                         format!(
131                             "Account {} has unregisterd. unlocking token goes to contract.",
132                             sender_id
133                         )
134                         .as_bytes(),
135                     );
136                 }
137             }
138         };
139     }
```

Listing 2.2: xref-token/src/xref.rs

```
/// unstake token and send assets back to the predecessor account.
/// Requirements:
/// * The predecessor account should be registered.
/// * amount must be a positive integer.
/// * The predecessor account should have at least the amount of tokens.
/// * Requires attached deposit of exactly 1 yoctoNEAR.
#[payable]
pub fn unstake(&mut self, amount: U128) -> Promise {
        // Checkpoint
        self.distribute_reward();

        assert_one_yocto();
        let account_id = env::predecessor_account_id();
        let amount: Balance = amount.into();

        assert!(self.ft.total_supply > 0, "ERR_EMPTY_TOTAL_SUPPLY");
        let unlocked = (U256::from(amount) * U256::from(self.locked_token_amount) / U256::from(self
            .ft.total_supply)).as_u128();

        self.ft.internal_withdraw(&account_id, amount);
        assert!(self.ft.total_supply >= 10u128.pow(18), "ERR_KEEP_AT_LEAST_ONE_XREF");
        self.locked_token_amount -= unlocked;

        log!("Withdraw {} NEAR from {}", amount, account_id);

        ext_fungible_token::ft_transfer(
            account_id.clone(),
            U128(unlocked),
            None,
            &self.locked_token,
            1,
            GAS_FOR_FT_TRANSFER,
        )
        .then(ext_self::callback_post_unstake(
            account_id.clone(),
            U128(unlocked),
            U128(amount),
            &env::current_account_id(),
            NO_DEPOSIT,
            GAS_FOR_RESOLVE_TRANSFER,
        ))
    }
```

Listing 2.3: xref-token/src/xref.rs

Impact The unstaked tokens of the unregistered account may be lost.

Suggestion If the sender's account doesn't exist, record the amounts of tokens in the owner's account as `lostfound`.

Feedback from the Project Will improve the event content to include the amount in next contract upgrade.

2.1.3 Optional Reward Distribution before Modifying Reward Rate

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The owner can modify the reward rate of `xREF` with the function `modify_reward_per_sec()`. However, before updating to the new reward rate, the owner can decide whether or not to distribute rewards with the original reward rate (lines 20-22).

```
18    pub fn modify_reward_per_sec(&mut self, reward_per_sec: U128, distribute_before_change: bool)
          {
19        self.assert_owner();
20        if distribute_before_change {
21            self.distribute_reward();
22        }
23        self.reward_per_sec = reward_per_sec.into();
24    }
```

Listing 2.4: xref-token/src/owner.rs

Impact Users may receive an unexpected amount of rewards, which is unfair.

Suggestion Trigger the distribution whenever the `Contract.reward_per_sec` is modified.

Feedback from the Project Will remove that `bool` argument and force the logic to distribute reward before changing `reward_per_sec`.

2.2 Additional Recommendation

2.2.1 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 implementation of the contract, 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 this smart contract (as a parameter of function `ft_on_transfer()`) and the actual number of transferred tokens (the token smart contract itself). That's because a small number of tokens will be burned by the token smart contract.

Suggestion I Do not set `Contract.locked_token` to an elastic supply token.

Feedback from the Project We will not set an elastic supply token as the `Contract.locked_token`.

2.2.2 Lack of Check on the Success of Upgrade

Status Confirmed

Introduced by `Version 1`

Description In function `upgrade()`, the entire upgrade process consists of the deployment of the contract code and the migration of the contract state. However, if any of the processes do not execute as expected, the `Contract.owner_id` may not be able to invoke any privileged functions, leading to a potential `DoS` problem.

```
/// 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 extern "C" fn upgrade() {
    env::setup_panic_hook();
    env::set_blockchain_interface(Box::new(near_blockchain::NearBlockchain {}));
    let contract: Contract = env::state_read().expect("ERR_CONTRACT_IS_NOT_INITIALIZED");
    contract.assert_owner();
    let current_id = env::current_account_id().into_bytes();
    let method_name = "migrate".as_bytes().to_vec();
    unsafe {
        BLOCKCHAIN_INTERFACE.with(|b| {
            // Load input into register 0.
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .input(0);
            let promise_id = b
                .borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_create(current_id.len() as _, current_id.as_ptr() as _);
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_action_deploy_contract(promise_id, u64::MAX as _, 0);
            let attached_gas = env::prepaid_gas() - env::used_gas() - GAS_FOR_MIGRATE_CALL;
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_action_function_call(
                    promise_id,
                    method_name.len() as _,
                    method_name.as_ptr() as _,
                    0 as _,
                    0 as _,
                    0 as _,
                    attached_gas,
                );
        });
    }
}
```

Listing 2.5: xref-token/src/owner.rs

Suggestion I Invoke the view function `contract_metadata()` to return the contract's basic information to ensure the state migrates successfully.

```
/// Return contract basic info
pub fn contract_metadata(&self) -> ContractMetadata {
    let to_be_distributed =
        self.try_distribute_reward(nano_to_sec(env::block_timestamp()));
    ContractMetadata {
        version: env!("CARGO_PKG_VERSION").to_string(),
        owner_id: self.owner_id.clone(),
        locked_token: self.locked_token.clone(),
        undistributed_reward: self.undistributed_reward.into(),
        locked_token_amount: self.locked_token_amount.into(),
        cur_undistributed_reward: (self.undistributed_reward - to_be_distributed).into(),
        cur_locked_token_amount: (self.locked_token_amount + to_be_distributed).into(),
        supply: self.ft.total_supply.into(),
        prev_distribution_time_in_sec: self.prev_distribution_time_in_sec,
        reward_genesis_time_in_sec: self.reward_genesis_time_in_sec,
        reward_per_sec: self.reward_per_sec.into(),
        account_number: self.account_number,
    }
}
```

Listing 2.6: xref-token/src/views.rs

Feedback from the Project Will add this logic in the next contract upgrade.

2.2.3 Lack of Check on the Gas Used by migrate()

Status Confirmed

Introduced by `Version 1`

Description There is no check on whether the `attached_gas` is enough for function `migrate()`.

```
/// 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 extern "C" fn upgrade() {
    env::setup_panic_hook();
    env::set_blockchain_interface(Box::new(near_blockchain::NearBlockchain {}));
    let contract: Contract = env::state_read().expect("ERR_CONTRACT_IS_NOT_INITIALIZED");
    contract.assert_owner();
    let current_id = env::current_account_id().into_bytes();
    let method_name = "migrate".as_bytes().to_vec();
    unsafe {
        BLOCKCHAIN_INTERFACE.with(|b| {
            // Load input into register 0.
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .input(0);
            let promise_id = b
                .borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_create(current_id.len() as _, current_id.as_ptr() as _);
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_action_deploy_contract(promise_id, u64::MAX as _, 0);
            let attached_gas = env::prepaid_gas() - env::used_gas() - GAS_FOR_MIGRATE_CALL;
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_action_function_call(
                        promise_id,
                        method_name.len() as _,
                        method_name.as_ptr() as _,
                        0 as _,
                        0 as _,
                        0 as _,
                        attached_gas,
                    );
            });
        }
    }
```

Listing 2.7: xref-token/src/owner.rs

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

Feedback from the Project Will add this logic in the next contract upgrade.

2.2.4 Improper Log Emission

Status Confirmed

Introduced by `Version 1`

Description The withdrawn token in function `unstake()` is `xREF` while the log in line 80 indicates the token is `NEAR`.

```
64      #[payable]
65      pub fn unstake(&mut self, amount: U128) -> Promise {
66          // Checkpoint
67          self.distribute_reward();
68
69          assert_one_yocto();
70          let account_id = env::predecessor_account_id();
71          let amount: Balance = amount.into();
72
73          assert!(self.ft.total_supply > 0, "ERR_EMPTY_TOTAL_SUPPLY");
74          let unlocked = (U256::from(amount) * U256::from(self.locked_token_amount) / U256::from(self
                .ft.total_supply)).as_u128();
75
76          self.ft.internal_withdraw(&account_id, amount);
77          assert!(self.ft.total_supply >= 10u128.pow(18), "ERR_KEEP_AT_LEAST_ONE_XREF");
78          self.locked_token_amount -= unlocked;
79
80          log!("Withdraw {} NEAR from {}", amount, account_id);
81
82          ext_fungible_token::ft_transfer(
83              account_id.clone(),
84              U128(unlocked),
85              None,
86              &self.locked_token,
87              1,
88              GAS_FOR_FT_TRANSFER,
89          )
90          .then(ext_self::callback_post_unstake(
            account_id.clone(),
            U128(unlocked),
            U128(amount),
            &env::current_account_id(),
            NO_DEPOSIT,
            GAS_FOR_RESOLVE_TRANSFER,
        ))
    }
```

Listing 2.8: xref-token/src/xref.rs

Suggestion I Change `"NEAR"` into `"xREF"` in line 80 of function `unstake()`.

Feedback from the Project Will fix this log in the next contract upgrade.

2.3 Notes

2.3.1 Assumption on the Secure Implementation of Dependencies

Status Confirmed

Introduced by `Version 1`

Description The `XRef_Token_Contract` is built based on the crates `NEAR-SDK` (version 3.1.0) and `near-contract-standards` (version 3.1.0).

```
/*!
* XRef NEP-141 Token contract
*
*/
use near_contract_standards::fungible_token::metadata::{
    FungibleTokenMetadata, FungibleTokenMetadataProvider, FT_METADATA_SPEC,
};
use near_contract_standards::fungible_token::FungibleToken;
```

Listing 2.9: xref-token/src/lib.rs

```
near_contract_standards::impl_fungible_token_core!(Contract, ft);
```

Listing 2.10: xref-token/src/lib.rs

```
use near_contract_standards::storage_management::{
    StorageBalance, StorageBalanceBounds, StorageManagement,
};
```

Listing 2.11: xref-token/src/storage_impl.rs

The required interfaces and the basic functionality listed below are provided in the contract:

* `NEP-141` (Fungible Token Standard)
* `NEP-145` (Storage Management Standard)
* `NEP-148` (Fungible Token Metadata Standard)

In this audit, we assume the standard library provided by `NEAR-SDK-RS` [1] (i.e., `near_contract_standards`) has no security issues.

[1] https://github.com/near/near-sdk-rs

2.3.2 Assumption on the Secure Management of the DAO

Status Confirmed

Introduced by `Version 1`

Description The `Contract.owner_id` has the privilege to configure some of the system parameters (e.g., `Contract.reward_per_sec` and `Contract.reward_genesis_time_in_sec` and upgrade the contract.

According to the contract deployment on NEAR's `mainnet`, a public `DAO` named `ref-finance.sputnik-dao.near` is taking ownership of this contract `xtoken.ref-finance.near`. This audit assumes that the public `DAO` is managed normally without security issues.