BLOCKSEC

Security Audit Report for Token Locker

Date: Jul 08, 2024 **Version:** 2.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Ref Finance
Target	Token Locker

Version History

Version	Date	Description
Version1	May 07, 2024	First release
Version2	Jul 08, 2024	Second release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-07-08 12:24:04+0000

About BlockSec 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 14 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 target of this audit is the code repository of Token Locker[1] of Ref Finance.

In the Token Locker, users can lock tokens in the contract and unlock them from the project after a certain period of time.

The auditing process is iterative. Specifically, we would 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 table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Token Locker	`Version 1`	`1da97ead5c0d9243aba5a129b165e713b4663efb`
	`Version 2`	`cc14cff5bff449bbf0777462c1f1a738a8898fcd`
	`Version 3`	`2d78d7cb9d8ffe370fbafcc4bb61f411aa628027`

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] https://github.com/ref-finance/token_locker

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

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

Furthermore, the status of a discovered 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 **four** potential issue. Besides, we also have **one** recommendation.

- Medium Risk: 4
- Recommendation: 1

ID	Severity	Description	Category	Status
1	Medium	Lack of token whitelist check	DeFi Security	Fixed
2	Medium	Potential incorrect parsing of mft token id	DeFi Security	Fixed
3	Medium	Lack of charging `NEAR` in function withdraw	DeFi Security	Fixed
4	Medium	Incorrect update on `unlock_time_sec` in function `after_token_burn()`	DeFi Security	Fixed
5	-	Incorrect error message	Recommendation	Fixed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of token whitelist check

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the protocol design, users can lock up to 64 different tokens. However, as there is no validation in function `ft_on_transfer()` to ensure that the tokens being locked comply with the `NEP-141` standard, malicious users are able to lock tokens for others, thereby hindering users from locking their tokens.

```
58 pub fn add_lock(&mut self, token_id: &String, amount: U128, unlock_time_sec: u32) {
59   require!(
60       self.locked_tokens.len() < MAX_LOCK_NUM,
61       "Exceed MAX_LOCK_NUM"
62   );
63   require!(
64       nano_to_sec(env::block_timestamp()) < unlock_time_sec,
65       "Invalid unlock_time_sec"
66   );
67   self.locked_tokens.insert(
68       token_id.clone(),
69       LockInfo {
70           locked_balance: amount,
71           unlock_time_sec,
72       },
73   );
74}
```

Listing 2.1: accounts.rs

```rust
fn ft_on_transfer(
    &mut self,
    sender_id: AccountId,
    amount: U128,
    msg: String,
) -> PromiseOrValue<U128> {
    let token_id = env::predecessor_account_id();
    let mut account = self.internal_unwrap_account(&sender_id);


    let message = serde_json::from_str::<TokenReceiverMessage>(&msg).expect("INVALID MSG");
    match message {
        TokenReceiverMessage::Lock { unlock_time_sec } => {
            if let Some(lock_info) = account.locked_tokens.get_mut(&token_id.to_string()) {
                lock_info.append_lock(amount, unlock_time_sec);
                Event::AppendToken {
                    account_id: &sender_id,
                    token_id: &token_id.to_string(),
                    amount: &amount,
                    unlock_time_sec,
                }
                .emit();
            } else {
                account.add_lock(&token_id.to_string(), amount, unlock_time_sec);
                Event::LockedToken {
                    account_id: &sender_id,
                    token_id: &token_id.to_string(),
                    amount: &amount,
                    unlock_time_sec,
                }
                .emit();
            }
        }
    }
    self.internal_set_account(&sender_id, account);
    PromiseOrValue::Value(U128(0))
}
```

Listing 2.2: token_receiver.rs

Impact Users will be unable to lock their tokens.

Suggestion Add token whitelist check.

2.1.2 Potential incorrect parsing of mft token id

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the locking process for `mft` tokens, the function `generate_mft_token_id()` concatenates the `mft` contract account id, the `MFT_TAG`(`@` character), and the `mft` token_id to create

the token_id for the deposited `mft` token. During withdrawal, the function `parse_token_id()` parses the recorded token_id back into the `mft` token to be sent.

However, function `parse_token_id()` uses `token_id.split(MFT_TAG).collect()` for parsing, if the user's previously entered mft token id includes the `@` character, the parsing process will treat the `@` in the mft token id as the `MFT_TAG` instead of part of the `mft` token id. This will result in the incorrect parsing of the previously locked `mft` token.

```
14  pub fn generate_mft_token_id(token_id: String) -> String {
15      format!("{}{}{}", env::predecessor_account_id(), MFT_TAG, token_id)
16  }
```

Listing 2.3: utils.rs

```
18  pub fn parse_token_id(token_id: &String) -> (AccountId, Option<String>) {
19      let v: Vec<&str> = token_id.split(MFT_TAG).collect();
20      if v.len() == 1 {
21          let contract_id: AccountId = v[0].parse().unwrap();
22          (contract_id, None)
23      } else if v.len() == 2 {
24          let contract_id: AccountId = v[0].parse().unwrap();
25          (contract_id, Some(v[1].to_string()))
26      } else {
27          env::panic_str("INVALID TOKEN ID");
28      }
29  }
```

Listing 2.4: utils.rs

```
61  fn mft_on_transfer(
62      &mut self,
63      token_id: String,
64      sender_id: AccountId,
65      amount: U128,
66      msg: String,
67  ) -> PromiseOrValue<U128> {
68      let token_id = generate_mft_token_id(token_id);
69      let mut account = self.internal_unwrap_account(&sender_id);
70
71
72      let message = serde_json::from_str::<TokenReceiverMessage>(&msg).expect("INVALID MSG");
73      match message {
74          TokenReceiverMessage::Lock { unlock_time_sec } => {
75              if let Some(lock_info) = account.locked_tokens.get_mut(&token_id) {
76                  lock_info.append_lock(amount, unlock_time_sec);
77                  Event::AppendToken {
78                      account_id: &sender_id,
79                      token_id: &token_id.to_string(),
80                      amount: &amount,
81                      unlock_time_sec,
82                  }
83                  .emit();
84              } else {
85                  account.add_lock(&token_id, amount, unlock_time_sec);
```

```
86                  Event::LockedToken {
87                      account_id: &sender_id,
88                      token_id: &token_id.to_string(),
89                      amount: &amount,
90                      unlock_time_sec,
91                  }
92                  .emit();
93              }
94          }
95      }
96      self.internal_set_account(&sender_id, account);
```

Listing 2.5: token_receiver.rs

```
177 pub fn transfer_token(&self, account_id: &AccountId, token_id: String, amount: U128) {
178     let (contract_id, mft_token_id) = parse_token_id(&token_id);
179     if let Some(mft_token_id) = mft_token_id {
180         ext_multi_fungible_token::ext(contract_id.clone())
181             .with_attached_deposit(NearToken::from_yoctonear(1))
182             .with_static_gas(GAS_FOR_TOKEN_TRANSFER)
183             .mft_transfer(mft_token_id, account_id.clone(), amount, None)
184             .then(
185                 Self::ext(env::current_account_id())
186                     .with_static_gas(GAS_FOR_AFTER_TOKEN_TRANSFER)
187                     .after_token_transfer(account_id.clone(), token_id.clone(), amount),
188             )
189     } else {
190         ext_fungible_token::ext(contract_id.clone())
191             .with_attached_deposit(NearToken::from_yoctonear(1))
192             .with_static_gas(GAS_FOR_TOKEN_TRANSFER)
193             .ft_transfer(account_id.clone(), amount, None)
194             .then(
195                 Self::ext(env::current_account_id())
196                     .with_static_gas(GAS_FOR_AFTER_TOKEN_TRANSFER)
197                     .after_token_transfer(account_id.clone(), token_id.to_string(), amount),
198             )
199     };
200 }
```

Listing 2.6: account.rs

Impact Users will be unable to withdraw their locked `mft` tokens.

Suggestion Revise the logic accordingly.

2.1.3 Lack of charging `NEAR` in function withdraw

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the token withdrawal process, the function `transfer_token()` requires attaching 1 `yocto NEAR` to send the token. However, the function `withdraw()` does not require users

to pay this portion of NEAR. The contract will cover it instead.

```
94  pub fn withdraw(&mut self, token_id: String, amount: Option<U128>) {
95      let account_id = env::predecessor_account_id();
96      let mut account = self.internal_unwrap_account(&account_id);
97
98
99      if let Some(mut lock_info) = account.locked_tokens.remove(&token_id) {
100         require!(
101             lock_info.unlock_time_sec <= nano_to_sec(env::block_timestamp()),
102             "Token still locked"
103         );
104         let amount = amount.unwrap_or(lock_info.locked_balance);
105         lock_info.locked_balance = U128(
106             lock_info
107                 .locked_balance
108                 .0
109                 .checked_sub(amount.0)
110                 .expect("Lock balance not enough"),
111         );
112         if lock_info.locked_balance.0 > 0 {
113             account.locked_tokens.insert(token_id.clone(), lock_info);
114         }
115         self.internal_set_account(&account_id, account);
116         self.transfer_token(&account_id, token_id.clone(), amount);
117         Event::WithdrawStarted {
118             account_id: &account_id,
119             token_id: &token_id,
120             amount: &amount,
121         }
122         .emit();
123     } else {
124         env::panic_str("Invalid token_id");
125     }
126 }
```

Listing 2.7: accounts.rs

```
177 pub fn transfer_token(&self, account_id: &AccountId, token_id: String, amount: U128) {
178     let (contract_id, mft_token_id) = parse_token_id(&token_id);
179     if let Some(mft_token_id) = mft_token_id {
180         ext_multi_fungible_token::ext(contract_id.clone())
181             .with_attached_deposit(NearToken::from_yoctonear(1))
182             .with_static_gas(GAS_FOR_TOKEN_TRANSFER)
183             .mft_transfer(mft_token_id, account_id.clone(), amount, None)
184             .then(
185                 Self::ext(env::current_account_id())
186                     .with_static_gas(GAS_FOR_AFTER_TOKEN_TRANSFER)
187                     .after_token_transfer(account_id.clone(), token_id.clone(), amount),
188             )
189     } else {
190         ext_fungible_token::ext(contract_id.clone())
191             .with_attached_deposit(NearToken::from_yoctonear(1))
192             .with_static_gas(GAS_FOR_TOKEN_TRANSFER)
```

```
193             .ft_transfer(account_id.clone(), amount, None)
194             .then(
195                 Self::ext(env::current_account_id())
196                     .with_static_gas(GAS_FOR_AFTER_TOKEN_TRANSFER)
197                     .after_token_transfer(account_id.clone(), token_id.to_string(), amount),
198             )
199     };
200 }
```

Listing 2.8: account.rs

Impact The contract may not have enough `NEAR` to transfer tokens.

Suggestion Add charge for `NEAR` in function `withdraw()`.

2.1.4 Incorrect update on `unlock_time_sec` in function `after_token_burn()`

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the `account.rs` file, users can remove their locked tokens with function `burn()` and transfer the corresponding tokens to the `burn_account_id`. In the callback function `after_token_burn()`, if the token transfer fails, it will rollback the user's lockup record. Specifically, when all the locked tokens are burned, the `unlock_time_sec` is updated to the current timestamp. Which is incorrect.

```
131 pub fn burn(&mut self, token_id: String, amount: Option<U128>) {
132     assert_one_yocto();
133     let account_id = env::predecessor_account_id();
134     let mut account = self.internal_unwrap_account(&account_id);
135
136
137     if let Some(mut lock_info) = account.locked_tokens.remove(&token_id) {
138         let amount = amount.unwrap_or(lock_info.locked_balance);
139         lock_info.locked_balance = U128(
140             lock_info
141                 .locked_balance
142                 .0
143                 .checked_sub(amount.0)
144                 .expect("Lock balance not enough"),
145         );
146         if lock_info.locked_balance.0 > 0 {
147             account.locked_tokens.insert(token_id.clone(), lock_info);
148         }
149         self.internal_set_account(&account_id, account);
150         self.burn_token(&account_id, token_id.clone(), amount);
151         Event::BurnStarted {
152             account_id: &account_id,
153             token_id: &token_id,
154             amount: &amount,
155         }
```

```
156             .emit();
157         } else {
158             env::panic_str("Invalid token_id");
159         }
160     }
```

Listing 2.9: account.rs

```
284     pub fn burn_token(&self, account_id: &AccountId, token_id: String, amount: U128) {
285         let burn_account_id = self.data().burn_account_id.clone().expect("Missing burn_account_id")
                ;
286         let (contract_id, mft_token_id) = parse_token_id(&token_id);
287         if let Some(mft_token_id) = mft_token_id {
288             ext_multi_fungible_token::ext(contract_id.clone())
289                 .with_attached_deposit(NearToken::from_yoctonear(1))
290                 .with_static_gas(GAS_FOR_TOKEN_TRANSFER)
291                 .mft_transfer(mft_token_id, burn_account_id, amount, None)
292                 .then(
293                     Self::ext(env::current_account_id())
294                         .with_static_gas(GAS_FOR_AFTER_TOKEN_BURN)
295                         .after_token_burn(account_id.clone(), token_id.clone(), amount),
296                 )
297         } else {
298             ext_fungible_token::ext(contract_id.clone())
299                 .with_attached_deposit(NearToken::from_yoctonear(1))
300                 .with_static_gas(GAS_FOR_TOKEN_TRANSFER)
301                 .ft_transfer(burn_account_id, amount, None)
302                 .then(
303                     Self::ext(env::current_account_id())
304                         .with_static_gas(GAS_FOR_AFTER_TOKEN_BURN)
305                         .after_token_burn(account_id.clone(), token_id.to_string(), amount),
306                 )
307         };
308     }
```

Listing 2.10: account.rs

```
210     pub fn after_token_burn(
211         &mut self,
212         account_id: AccountId,
213         token_id: String,
214         amount: U128,
215     ) -> bool {
216         let promise_success = is_promise_success();
217         if !promise_success {
218             if let Some(mut account) = self.internal_get_account(&account_id) {
219                 if let Some(lock_info) = account.locked_tokens.get_mut(&token_id.to_string()) {
220                     lock_info.locked_balance = U128(lock_info.locked_balance.0 + amount.0);
221                 } else {
222                     account.locked_tokens.insert(
223                         token_id.clone(),
224                         LockInfo {
225                             locked_balance: amount,
```

```
226                         unlock_time_sec: nano_to_sec(env::block_timestamp()),
227                     },
228                 );
229             }
230             self.internal_set_account(&account_id, account);
231             Event::BurnFailed {
232                 account_id: &account_id,
233                 token_id: &token_id,
234                 amount: &amount,
235             }
236             .emit();
237         } else {
238             Event::BurnLostfound {
239                 account_id: &account_id,
240                 token_id: &token_id,
241                 amount: &amount,
242             }
243             .emit();
244         }
245     } else {
246         Event::BurnSucceeded {
247             account_id: &account_id,
248             token_id: &token_id,
249             amount: &amount,
250         }
251         .emit();
252 
253     }
254     promise_success
255 }
```

Listing 2.11: account.rs

Impact In this case, users can invoke the function `withdraw()` to withdraw assets and bypass the `unlock_time_sec` check.

Suggestion Revise the logic to ensure that the state correctly rolls back to the user's previous lockup status.

2.2 Additional Recommendation

2.2.1 Incorrect error message

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `storage_deposit()`, the error message `insufficient depost` is incorrect.

```
12 fn storage_deposit(
13   &mut self,
14   account_id: Option<AccountId>,
```

```
    registration_only: Option<bool>,
) -> StorageBalance {
    let amount = env::attached_deposit();
    let account_id = account_id.unwrap_or_else(|| env::predecessor_account_id());
    let already_registered = self.internal_get_account(&account_id).is_some();
    if amount < STORAGE_BALANCE_MIN_BOUND && !already_registered {
        env::panic_str("Insufficient depost");
    }


    if already_registered {
        if !amount.is_zero() {
            Promise::new(env::predecessor_account_id()).transfer(amount);
        }
    } else {
        self.internal_set_account(&account_id, Account::new(&account_id).into());
        let refund = amount.checked_sub(STORAGE_BALANCE_MIN_BOUND).unwrap();
        if !refund.is_zero() {
            Promise::new(env::predecessor_account_id()).transfer(refund);
        }
        Event::AccountRegister { account_id: &account_id }.emit();
    }
    self.storage_balance_of(account_id).unwrap()
}
```

Listing 2.12: storage.sol

Suggestion Replace error message `insufficient depost` with `insufficient deposit`.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS