Security Audit Report for ram.defi

Date: Dec 25, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by BlockSec
Audit Team
Date: 2023.12.25 19:29:37
+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 . . . 2
1.3.4 Additional Recommendation . . . 3
1.4 Security Model . . . 3

2 Findings **4**

2.1 Software Security . . . 4
2.1.1 Potential incorrect destination in token issuance . . . 4
2.1.2 DoS due to rejected token transfer . . . 5
2.2 Additional Recommendation . . . 6
2.2.1 Prevent further token creation . . . 6
2.3 Notes . . . 7
2.3.1 Potential centralization risk . . . 7

Report Manifest

Item	Description
Client	Defibox
Target	ram.defi

Version History

Version	Date	Description
1.0	Dec 25, 2023	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

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

This audit targets the `ram.defi` contract for Defibox on the EOS blockchain. `ram.defi` allows users to buy RAM through the contract and issues `BRAM` tokens to the buyers. `BRAM` token holders are permitted to burn their tokens and sell RAM in exchange for EOS tokens.

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.

Version 1: File	MD5 Hash
ram.defi.cpp	`535bc43a200bde483dad78d329c08a03`
ram.defi.hpp	`768097e0a0814a2e09fd13cf1d613f97`
Version 2: File	MD5 Hash
ram.defi.cpp	`535d1b60201a8ad1aa7b01c357ab96cd`
ram.defi.hpp	`768097e0a0814a2e09fd13cf1d613f97`

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 C++ language and the compiler), the underlying toolchain (e.g., the EOS blockchain and the EOSIO contracts) 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 [1] and Common Weakness Enumeration [2]. 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.

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

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

Chapter 2 Findings

In total, we find **two** potential issues. Besides, we also have **one** recommendation and **one** note.

- High Risk: 1
- Low Risk: 1
- Recommendation: 1
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Potential incorrect destination in token issuance	Software Security	Fixed
2	High	DoS due to rejected token transfer	Software Security	Fixed
3	-	Prevent further token creation	Recommendation	Confirmed
4	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential incorrect destination in token issuance

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `ram.defi` contract, the buyer transfers EOS to the contract, then the contract buys RAM according to the value transferred, and issue corresponding amount of `BRAM` token to the buyer. In the `issue` function, the `BRAM` tokens are first issued to the `issuer`, and then transferred from the contract itself to the target `to` address. Therefore, the `issuer` must be set to the contract itself, otherwise the issue would fail due to insufficient balance. However, there is no check for this assumption upon token creation.

```
176   void ram::issue(const name& to, const asset& quantity) {
177       auto sym = quantity.symbol;
178       stats statstable(_self, sym.code().raw());
179       auto existing = statstable.find(sym.code().raw());
180       check(existing != statstable.end(), "token with symbol does not exist, create token before
              issue");
181       const auto& st = *existing;
182
183       check(quantity.amount <= st.max_supply.amount - st.supply.amount, "quantity exceeds
              available supply");
184
185       statstable.modify(st, same_payer, [&](auto& s) { s.supply += quantity; });
186
187       add_balance(st.issuer, quantity, st.issuer);
188       if (to != st.issuer) {
189           ram::transfer_action transfer_act(get_self(), {get_self(), "active"_n});
190           transfer_act.send(get_self(), to, quantity, "issue");
```

```
191        }
192    }
```

Listing 2.1: ram.defi.cpp

In the following code segment, the `create` function is used to create new tokens (including the `BRAM` token). There is no restrictions on the `issuer` address.

```
 8void ram::create(const name& issuer, const asset& maximum_supply) {
 9    require_auth(get_self());
10
11    auto sym = maximum_supply.symbol;
12    check(sym.is_valid(), "invalid symbol name");
13    check(maximum_supply.is_valid(), "invalid supply");
14    check(maximum_supply.amount > 0, "max-supply must be positive");
15
16    stats statstable(get_self(), sym.code().raw());
17    auto existing = statstable.find(sym.code().raw());
18    check(existing == statstable.end(), "token with symbol already exists");
19
20    statstable.emplace(get_self(), [&](auto& s) {
21        s.supply.symbol = maximum_supply.symbol;
22        s.max_supply = maximum_supply;
23        s.issuer = issuer;
24    });
25}
```

Listing 2.2: ram.defi.cpp

Impact The token issue can fail due to incorrect logic.

Suggestion Fix the logic in the issue function.

2.1.2 DoS due to rejected token transfer

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `ram.defi` contract, it is allowed for users to transfer the `BRAM` token to the contract, which would be treated as burning the `BRAM` token and selling the RAM previously bought. However, there is a flawed logic in the implementation of the transfer. In the following code segment, the call to `require_recipient` function triggers a logic of notification to the `to` address, which might be the contract itself.

```
27    [[eosio::action]]
28    void ram::transfer(const name& from, const name& to, const asset& quantity, const string& memo
          ) {
29        check(from != to, "cannot transfer to self");
30        require_auth(from);
31        check(is_account(to), "to account does not exist");
32
33        auto sym = quantity.symbol.code();
```

```
34        stats statstable(get_self(), sym.raw());
35        const auto& st = statstable.get(sym.raw());
36
37        require_recipient(from);
38        require_recipient(to);
39
40        check(quantity.is_valid(), "invalid quantity");
41        check(quantity.amount > 0, "must transfer positive quantity");
42        check(quantity.symbol == st.supply.symbol, "symbol precision mismatch");
43        check(memo.size() <= 256, "memo has more than 256 bytes");
44
45        if (to == get_self()) {
46            do_withdraw(from, quantity);
```

Listing 2.3: ram.defi.cpp

However, in the following code segment to answer the notification, the function `on_transfer` only allows the transfer of the `EOS` token to proceed with the transfer.

```
27    [[eosio::on_notify("*::transfer")]]
28    void ram::on_transfer(const name& from, const name& to, const asset& quantity, const string&
          memo) {
29        // authenticate incoming `from` account
30        require_auth(from);
31
32        // ignore transfers
33        if (to != get_self())
34            return;
35
36        const name contract = get_first_receiver();
37        if (contract != EOS_CONTRACT || quantity.symbol != EOS)
38            check(false, "ram.defi::deposit: only transfer [eosio.token/EOS] ");
39
40        do_deposit(from, quantity);
41    }
```

Listing 2.4: ram.defi.cpp

Impact Withdrawal through transferring `BRAM` tokens to the contract itself would not succeed.

Suggestion Move the notification logic to another branch to prevent notifying the contract itself.

2.2 Additional Recommendation

2.2.1 Prevent further token creation

Status Confirmed

Introduced by `Version 1`

Description The `ram.defi` contract is forked from the original official `eosio.token` contract, and preserves the `create` function that allows privileged accounts to create new tokens. However, in the `ram.defi` contract, only `BRAM` token is used, so no new tokens should be allowed to be created after the creation of the `BRAM` token.

```
 8   void ram::create(const name& issuer, const asset& maximum_supply) {
 9       require_auth(get_self());
10
11       auto sym = maximum_supply.symbol;
12       check(sym.is_valid(), "invalid symbol name");
13       check(maximum_supply.is_valid(), "invalid supply");
14       check(maximum_supply.amount > 0, "max-supply must be positive");
15
16       stats statstable(get_self(), sym.code().raw());
17       auto existing = statstable.find(sym.code().raw());
18       check(existing == statstable.end(), "token with symbol already exists");
19
20       statstable.emplace(get_self(), [&](auto& s) {
21           s.supply.symbol = maximum_supply.symbol;
22           s.max_supply = maximum_supply;
23           s.issuer = issuer;
24       });
25   }
```

Listing 2.5: ram.defi.cpp

Suggestion Disable further token creation after the `BRAM` token has been created.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the `ram.defi` contract, an admin account is allowed to disable the deposit and withdrawal functionality through privileged functions. Specifically, the `updatestatus` function enables the admin to prevent all users from withdrawing. Although this function may be useful under extreme conditions, it does impose centralization risks on the users.

```
58   [[eosio::action]]
59   void ram::updatestatus(bool disabled_deposit, bool disabled_withdraw) {
60       require_auth(ADMIN_ACCOUNT);
61       ram::config_table _config(get_self(), get_self().value);
62       config_row config = _config.get_or_default();
63
64       config.disabled_deposit = disabled_deposit;
65       config.disabled_withdraw = disabled_withdraw;
66       _config.set(config, get_self());
67   }
```

Listing 2.6: ram.defi.cpp

Feedback from the Project The project maintains admin account through a multisig wallet, and ensure that no new token will be created after the creation of the `BRAM` token.