Security Audit
Report for ramx.eos

Date: September 4, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	RAMS
Target	ramx.eos

Version History

Version	Date	Description
1.0	September 4, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-09-04 12:52:26+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	C++
Approach	Semi-automatic and manual verification

The focus of this audit is the ramx.eos contract of RAMS. Specifically, the core `ramx.eos` contract allows users to trade deposited RAM directly, without requiring withdrawal from the `rambank.eos` contract.

It is important to note that only the C++ source files listed in the table below are included in the scope of this audit. Furthermore, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and therefore, are not included in the audit scope.

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.

Source	Version	File	MD5 Hash
ramx	Version 1	`ramx.eos/ramx.eos.cpp`	`a283a3313ff57cafcd55e8d19ac82884`
		`ramx.eos/ramx.eos.hpp`	`d1752fe88b29fe7cbe96bbb5246bcf6a`
		`internal/defines.hpp`	`64118e987354fc233bb111df2901533b`
		`internal/safemath.hpp`	`c93a58c712edc2399da594c16b01d308`
		`internal/utils.hpp`	`2318f740a64acf5d3f6264f2779b6404`
	Version 2	`ramx.eos/ramx.eos.cpp`	`37023c273de5b1050d04a654bea56379`
		`ramx.eos/ramx.eos.hpp`	`ea886e2784943a78805692a413608bde`
		`internal/defines.hpp`	`64118e987354fc233bb111df2901533b`
		`internal/safemath.hpp`	`c93a58c712edc2399da594c16b01d308`
		`internal/utils.hpp`	`2318f740a64acf5d3f6264f2779b6404`

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), the underlying compiling toolchain and the computing infrastructure (e.g., the blockchain runtime and system contracts of the EOS network) 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 (Likelihood)	Low (Likelihood)
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.

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

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

Furthermore, the status of a discovered item will fall into one of the following four categories:

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

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **one** recommendation and **one** note.

- High Risk: 2
- Recommendation: 1
- Note: 1

ID	Severity	Description	Category	Status
1	High	Potential DoS in sell order creation	Software Security	Fixed
2	High	Incorrect calculation in sell order processing	Software Security	Fixed
3	-	Remove duplicate checks	Recommendation	Fixed
4	-	Pontential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential DoS in sell order creation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `sellorder` function, the `price` parameter is passed without additional upper bound checks. The `amount` variable, calculated by multiplying the `price` with the `bytes` of RAM to be sold, represents the required amount of `EOS` tokens to fulfill the order. Since the `price` parameter lacks an upper bound check, the `amount` variable can be manipulated. This logic is reflected in Lines 47–61 of the following code segment.

In the `ramx.eos` contract, key statistics are recorded in the `stats` state variable. Specifically, Lines 80–84 of the code segment record the corresponding order quantity (of the `asset` type) and bytes of RAM (of `uint64_t` type). However, because the `asset` type includes overflow checks, a malicious actor could create an order and increase the `sell_quantity` close to the upper bound. This would cause all subsequent sell order creations to fail due to these overflow checks.

```
43  [[eosio::action]]
44  void ramx::sellorder(const name& owner, const uint64_t price, const uint64_t bytes) {
45      require_auth(owner);
46
47      auto config = _config.get();
48
49      check(price > 0, "ramx.eos::sellorder: price must be greater than 0");
50      check(bytes > 0, "ramx.eos::sellorder: bytes must be greater than 0");
```

```
    check(!config.disabled_pending_order, "ramx.eos::sellorder: pending order has been suspended");

    auto amount = uint128_t(price) * bytes / PRICE_PRECISION;
    check(amount <= asset::max_amount, "ramx.eos::sellorder: trade quantity too large");

    auto quantity = asset(amount, EOS);

    check(bytes >= config.min_trade_bytes,
         "ramx.eos::sellorder: bytes must be greater than " + std::to_string(config.min_trade_bytes));
    check(quantity >= config.min_trade_amount,
         "ramx.eos::sellorder: (price * bytes) must be greater than " + config.min_trade_amount.to_string());

    // freeze
    bank::freeze_action freeze(RAM_BANK_CONTRACT, {get_self(), "active"_n});
    freeze.send(owner, bytes);

    // order
    auto order_id = next_order_id();
    _order.emplace(get_self(), [&](auto& row) {
       row.id = order_id;
       row.type = ORDER_TYPE_SELL;
       row.owner = owner;
       row.price = price;
       row.bytes = bytes;
       row.quantity = quantity;
       row.created_at = current_time_point();
    });

    // update stat
    auto stat = _stat.get_or_default();
    stat.num_sell_orders += 1;
    stat.sell_quantity += quantity;
    stat.sell_bytes += bytes;
    _stat.set(stat, get_self());
```

Listing 2.1: contracts/ramx.eos/ramx.eos.cpp

Impact May cause DoS in the sell order creation process.

Suggestion Implement upper bound checks for the parameters.

2.1.2 Incorrect calculation in sell order processing

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `sell` function, the `remain_bytes` variable represents the remaining bytes of RAM available to the user in the `rambank.eos` contract for order matching. However, there are two issues in handling this variable:

1. The `remain_bytes` is initiated by reading the `bytes` field from the `deposit_table` in the `rambank.eos` contract. However, this does not account for RAM that has been frozen for order manipulation, which is not considered in the calculation.
2. After processing each order, the `remain_bytes` varibale is not updated correctly to reflect the user's current remaining RAM.

```
[[eosio::action]]
ramx::trade_result ramx::sell(const name& owner, const vector<uint64_t>& order_ids) {
    require_auth(owner);

    check(order_ids.size() > 0, "ramx.eos::sell: order_ids cannot be empty");
    check(!has_duplicate(order_ids), "ramx.eos::sell: invalid duplicate order_ids");

    auto config = _config.get();
    check(!config.disabled_trade, "ramx.eos::sell: trade has been suspended");

    auto stat = _stat.get_or_default();

    bank::deposit_table _deposit(RAM_BANK_CONTRACT, RAM_BANK_CONTRACT.value);
    auto deposit_itr = _deposit.require_find(owner.value, "ramx.eos::sell: no ram to sell");

    asset total_fees = {0, EOS};
    asset total_quantity = {0, EOS};
    uint64_t total_bytes = 0;
    vector<asset> fee_list;
    vector<uint64_t> trade_order_ids;
    uint64_t remain_bytes = deposit_itr->bytes;
    for (const auto& order_id : order_ids) {
        auto order_itr = _order.find(order_id);

        if (order_itr == _order.end() || order_itr->type != ORDER_TYPE_BUY || remain_bytes <
            order_itr->bytes) continue;

        // fees
        const auto fees = order_itr->quantity * config.fee_ratio / RATIO_PRECISION;

        // erase order
        _order.erase(order_itr);

        // transfer ram to buyer
        if (owner != order_itr->owner) {
            bank::transfer_action transfer(RAM_BANK_CONTRACT, {get_self(), "active"_n});
            transfer.send(owner, order_itr->owner, order_itr->bytes, "Sell RAMX");
        }

        total_fees += fees;
        total_quantity += order_itr->quantity;
        total_bytes += order_itr->bytes;
        fee_list.push_back(fees);
        trade_order_ids.push_back(order_id);
    }
```

Listing 2.2: contracts/ramx.eos/ramx.eos.cpp

Impact Incorrect calculation of sell orders can lead to unexpected failures during order processing.

Suggestion Refactor the sell order processing logic.

2.2 Additional Recommendation

2.2.1 Remove duplicate checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `cancelorder` function, duplicate checks can be removed.

```
check(!has_duplicate(order_ids), "ramx.eos::cancelorder: invalid duplicate order_ids");
check(!has_duplicate(order_ids), "ramx.eos::cancelorder: invalid duplicate order_ids");
```

Listing 2.3: contracts/ramx.eos/ramx.eos.cpp

Impact N/A

Suggestion Remove duplicate checks.

2.3 Note

2.3.1 Pontential centralization risks

Introduced by `Version 1`

Description The `ramx.eos` contract introduces mechanisms that can increase centralization risk. Specifically, key parameters can be adjusted by the project maintainers. For example, they can disable trading and order creation. If these parameters are set to incorrect or unexpected values, the contract may malfunction or cause users to lose funds.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS