BLOCKSEC

Security Audit Report for RAMS RAMBank

Date: June 29, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	RAMS
Target	RAMS RAMBank

Version History

Version	Date	Description
1.0	June 29, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-30 03:20:47+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 RAMS RAMBank. Specifically, the core contract (`rambank.eos`) allows the users to stake RAM to the contract and receives rent tokens as rewards. Conversely, the contract also allows privileged users to borrow and repay RAM from the contract, by depositing rent tokens as fees.

It is important to note that only the C++ source files under the 'contracts' directory are included in the scope of this audit. Furthermore, all the dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and therefore, they 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
RAMBank	Version 1	`rambank.eos/rambank.eos.cpp`	`18bd30f1aba1fe66ec11cc6f84cfbfc0`
		`rambank.eos/rambank.eos.hpp`	`23d131d9fdd2c5eeed495e98de5b102d`
		`internal/defines.hpp`	`64118e987354fc233bb111df2901533b`
		`internal/safemath.hpp`	`c93a58c712edc2399da594c16b01d308`
		`internal/utils.hpp`	`2318f740a64acf5d3f6264f2779b6404`
	Version 2	`rambank.eos/rambank.eos.cpp`	`9f64369462a812cdcf35655e92f674a8`
		`rambank.eos/rambank.eos.hpp`	`6adde52c689938187755e1630a858a62`
		`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*	*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.

[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** note.

- High Risk: 2
- Note: 1

ID	Severity	Description	Category	Status
1	High	Incorrect amount used for RAM deposit	Software Security	Fixed
2	High	Potential integer underflow	Software Security	Fixed
3	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Incorrect amount used for RAM deposit

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `rambank.eos` contract, the `do_deposit_ram` function is invoked when users transfer RAM to the contract. The function processes the user's deposit by first charging fees and then transferring the remaining RAM to the `RAM_CONTAINER` account. However, in the following code segment, the amount used for transferring the RAM to the `RAM_CONTAINER` account is incorrect. The amount of RAM after fees, i.e., the `to_bank` quantity, should be used instead of the original `bytes` quantity. In fact, the `to_bank` quantity is not used after its definition, which is incorrect according to the RAM deposit and withdrawal logic.

```
// issue stram
auto deposit_fee = bytes * config.deposit_fee_ratio / RATIO_PRECISION;
auto to_bank = bytes - deposit_fee;

// fees
if (deposit_fee > 0) {
    ram_transfer(get_self(), RAMFEES_EOS, deposit_fee, "deposit fee");
}
// transfer to ram container
ram_transfer(get_self(), RAM_CONTAINER, bytes, "deposit ram");
```

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

Impact The amount of RAM used for the deposit is incorrect, which can result in unexpected consequences.

Suggestion Refactor the calculation logic.

2.1.2 Potential integer underflow

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In the rambank.eos contract, users can withdraw their previously deposited RAM through the withdraw function. However, this function is vulnerable to an arithmetic underflow issue. The decrease of the row of the user in the deposit_table can underflow, causing the accounting to be incorrect. Additionally, the underflow would make the check for liquidity depletion ineffective.

```
[[eosio::action]]
void bank::withdraw(const name& owner, const uint64_t bytes) {
    require_auth(owner);

    check(bytes > 0, "rambank.eos::withdraw: cannot withdraw negative");
    bank::config_row config = _config.get_or_default();
    bank::stat_row stat = _stat.get_or_default();
    check(!config.disabled_withdraw, "rambank.eos::withdraw: withdraw has been suspended");
    check(config.usage_limit_ratio == 0
            || (stat.deposited_bytes - bytes > 0
               && stat.used_bytes * RATIO_PRECISION / (stat.deposited_bytes - bytes) < config.
                   usage_limit_ratio),
        "rambank.eos::withdraw: liquidity depletion");

    auto deposit_itr = _deposit.require_find(owner.value, "rambank.eos::withdraw: [deposits]
        does not exists");
    _deposit.modify(deposit_itr, same_payer, [&](auto& row) {
        row.bytes -= bytes;
    });
```

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

Impact A potential underflow vulnerability in the withdraw function can result in extraneous RAM that users can claim, which can cause significant loss to the users of the protocol.

Suggestion Add underflow checks for the withdrawal process.

2.2 Note

2.2.1 Pontential centralization risks

Description There are multiple mechanisms introduced in the rambank.eos contract that can increase the centralization risk:

1. Currently, borrowing is a privileged operation that only allows the protocol maintainer to invoke. It also requires no precondition (e.g., collaterals) to borrow arbitrary amount of RAM to arbitrary account. It is also not ensured in the contract that the borrower must repay or deposit rent tokens before borrowing.

2. The redemption of the deposited RAM cannot be withdrawn when the utilization ratio of the RAM (i.e., the total RAM borrowed of the total deposited) is higher than the `_todo` parameter. By setting this paramter to an unexpected value, none of the deposited RAM can be withdrawn, causing significant losses to the users.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS