# Security Audit Report for EOS REX/Tokenomics 2.0

**Date:** June 24, 2024 **Version:** 1.0

**Contact:** contact@blocksec.com

# Contents

**Chapter 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 . . . . . . . . . . 3
1.3.4 Additional Recommendation . . . . . . . . . . 3
1.4 Security Model . . . . . . . . . . 3

**Chapter 2 Findings** **5**

2.1 Software Security . . . . . . . . . . 5
2.1.1 Potential precision loss . . . . . . . . . . 5
2.1.2 Potential incosistent block producer count . . . . . . . . . . 6
2.2 Additional Recommendation . . . . . . . . . . 7
2.2.1 Check the validity of the total weight . . . . . . . . . . 7

## Report Manifest

| Item | Description |
|---|---|
| Client | EOS Network |
| Target | EOS REX/Tokenomics 2.0 |

## Version History

| Version | Date | Description |
|---|---|---|
| 1.0 | June 24, 2024 | First release |

## Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-07-02 09:15:58+0000

# 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 EOS REX/Tokenomics 2.0 of the EOS Network. The EOS ecosystem updates their system contracts via the pull request #150 [1] and the `eosio.reward` [2] repository.

It is important to note that only the C++ source files of the pull requests and the fee contract 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.

| Project | Version | Commit Hash |
|---|---|---|
| eosio.reward | Version 1 | d894de7caafc5384971886376acb44039454b71f |
| PR #150 | Version 1 | 04350f554d9dfa4e0568ee29c37f2b69f1298c18 |

## 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/eosnetworkfoundation/eos-system-contracts/pull/150

[2] https://github.com/eosnetworkfoundation/eosio.reward

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 [3] and Common Weakness Enumeration [4]. 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

|  |  | **Likelihood** |  |
|---|---|---|---|
|  |  | *High* | *Low* |
| **Impact** | *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.

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

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

- **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.

- Low Risk: 2
- Recommendation: 1

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | Low | Potential precision loss | Software Security | Acknowledged |
| 2 | Low | Potential incosistent block producer count | Software Security | Acknowledged |
| 3 | - | Check the validity of the total weight | Recommendation | Acknowledged |

The details are provided in the following sections.

## 2.1 Software Security

### 2.1.1 Potential precision loss

**Severity** Low

**Status** Acknowledged

**Introduced by** Version 1 (PR #150)

**Description** In the eosio.bpay contract, when calculating rewards for each block producer, the calculation (reward = quantity / producer_count) can potentially result in precision loss with low possibility. This may lead to some EOS tokens not being distributed correctly to the block producers, remaining instead in the eosio.bpay contract.

```
void bpay::on_transfer( const name from, const name to, const asset quantity, const string memo
    ) {
    if (from == get_self() || to != get_self()) {
        return;
    }

    // ignore eosio system incoming transfers (caused by bpay income transfers eosio => eosio.
        bpay => producer)
    if ( from == "eosio"_n) return;

    symbol system_symbol = eosiosystem::system_contract::get_core_symbol();

    check( quantity.symbol == system_symbol, "only core token allowed" );

    rewards_table _rewards( get_self(), get_self().value );
    eosiosystem::producers_table _producers( "eosio"_n, "eosio"_n.value );

    eosiosystem::global_state_singleton _global("eosio"_n, "eosio"_n.value);
    check( _global.exists(), "global state does not exist");
    uint16_t producer_count = _global.get().last_producer_schedule_size;

    asset reward = quantity / producer_count;
```

Listing 2.1: contracts/eosio.bpay/src/eosio.bpay.cpp

**Impact** Potential precision loss results in reduced payments to block producers and some EOS tokens being locked in the `eosio.bpay` contract.

**Suggestion** Refactor the payment calculation logic.

**Feedback from the Project** This is known, for the sake of reducing complexity, it wasn't worth adding "dust" calculations.

### 2.1.2 Potential incosistent block producer count

**Severity** Low

**Status** Acknowledged

**Introduced by** `Version 1` (PR #150)

**Description** When calculating rewards for block producers in the `eosio.bpay` contract, the `producer_count` is directly read from the global state, while the producers are read and filtered from the global producers array. There is a very low possibility that there are fewer efficient active block producers than the `producer_count`. In this case, the rewards to block producers can be incorrectly distributed.

```
eosiosystem::global_state_singleton _global("eosio"_n, "eosio"_n.value);
check( _global.exists(), "global state does not exist");
uint16_t producer_count = _global.get().last_producer_schedule_size;

asset reward = quantity / producer_count;

// get producer with the most votes
// using `by_votes` secondary index
auto idx = _producers.get_index<"prototalvote"_n>();
auto prod = idx.begin();

// get top n producers by vote, excluding inactive
std::vector<name> top_producers;
while (true) {
    if (prod == idx.end()) break;
    if (prod->is_active == false) continue;

    top_producers.push_back(prod->owner);

    if (top_producers.size() == producer_count) break;

    prod++;
}
```

Listing 2.2: contracts/eosio.bpay/src/eosio.bpay.cpp

**Impact** In rare circumstances, there may not be enough effective block producers (fewer than the global `producer_count`).

**Suggestion** Revise the corresponding code logic.

## 2.2 Additional Recommendation

### 2.2.1 Check the validity of the total weight

**Status** Acknowledged

**Introduced by** Version 1 (eosio.reward)

**Description** In the eosio.reward contract, it is recommended to check whether the total_weight is zero, as strategies can be changed and deleted.

```
[[eosio::action]]
void reward::distribute()
{
    // any authority is allowed to call this action

    strategies_table _strategies( get_self(), get_self().value );
    const uint32_t total_weight = get_total_weight();
```

**Listing 2.3:** eosio.reward.cpp

**Impact** The calculated total_weight can be zero.

**Suggestion** Add sanity checks.

**Feedback from the Project** I believe zero weight is still a valid value, this would mean no rewards are being distributed, otherwise we have no mechanism to stop the eosio.reward contract.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS