BLOCKSEC

Security Audit Report for EOS Wrapped RAM

Date: April 9, 2024 **Version:** 1.1

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 Additional Recommendation 5

2.1.1 Remove duplicated checks 5

2.2 Note 6

2.2.1 Potential centralization risks 6

Report Manifest

Item	Description
Client	EOS Network
Target	EOS Wrapped RAM

Version History

Version	Date	Description
1.0	April 4, 2024	First Release
1.1	April 9, 2024	Update status

Signature

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

Email: audit@blocksec.com

Date: 2024-04-09 10:52:33+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 EOS Wrapped RAM of the EOS Network [1] project. The EOS Wrapped RAM contract utilizes the RAM features from the system contracts of the EOS network to tokenize the RAM resource in the EOS network. It is important to note that only the C++ source files of the Wrapped RAM contract are included in the scope of this audit. Other files, including the source files in the `external` folder, are not within the scope of the audit. Additionally, please be aware that 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 scope of the audit.

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
EOS Wrapped RAM	Version 1	65dd6a950d3fb8b37b5ea81eedf4b84163aa6a0a
	Version 2	77b2121bc3d085aea59519ecc3f9d09987603887

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/eosio.wram

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 [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	*High*	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

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.

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

[3] 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 did not find potential security issues. However, there is **one** recommendation and **one** note.

- Recommendation: 1
- Note: 1

ID	Severity	Description	Category	Status
1	-	Remove duplicated checks	Recommendation	Fixed
2	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 Remove duplicated checks

Status Fixed in Version 2

Introduced by Version 1

Description There are duplicated checks in the implementation of the EOS Wrapped RAM contracts. For example, in the transfer function, if the WRAM tokens are transferred to the contract, the unwrap_ram logic is triggered to transfer the underlying RAM, represented by the WRAM tokens, back to the users.

```
void wram::transfer( const name& from,
                const name&  to,
                const asset& quantity,
                const string& memo )
{
    check( from != to, "cannot transfer to self" );
    require_auth( from );
    check( is_account( to ), "to account does not exist");
    auto sym = quantity.symbol.code();
    stats statstable( get_self(), sym.raw() );
    const auto& st = statstable.get( sym.raw() );

    require_recipient( from );
    require_recipient( to );

    check( quantity.is_valid(), "invalid quantity" );
    check( quantity.amount > 0, "must transfer positive quantity" );
    check( quantity.symbol == st.supply.symbol, "symbol precision mismatch" );
    check( memo.size() <= 256, "memo has more than 256 bytes" );

    auto payer = has_auth( to ) ? to : from;

    sub_balance( from, quantity );
    add_balance( to, quantity, payer );

    // user sends RAM token to contract
    // unwraps RAM, retires RAM token, and transfers RAM bytes to user
    // cannot use `on_notify` because contract cannot send inline action notifications to itself
    if ( to == get_self() ) unwrap_ram( from, quantity );

    // disable transfers to accounts on egress list
    check_disable_transfer( to );
}
```

Listing 2.1: src/token.cpp

In the `unwrap_ram` function, there are duplicated checks that have already been performed by the `transfer` function, as listed in the following code segment. It should be noted that the `unwrap_ram` function is only used within the `transfer` function.

```
void wram::unwrap_ram( const name to, const asset quantity )
{
  // update WRAM supply to reflect system RAM
  mirror_system_ram();

  // validate incoming token transfer
  check(quantity.symbol == RAM_SYMBOL, "Only the system " + RAM_SYMBOL.code().to_string() + " token is accepted for transfers.");
  check(quantity.amount > 0, "quantity must be positive" ); // shouldn't be possible
  check(is_account(to), "to account does not exist" ); // shouldn't be possible
  check(to != get_self(), "cannot transfer to self" ); // shouldn't be possible

  // ramtransfer to user
  eosiosystem::system_contract::ramtransfer_action ramtransfer_act{"eosio"_n, {get_self(), "active"_n}};
  ramtransfer_act.send(get_self(), to, quantity.amount, "unwrap ram");
}
```

Listing 2.2: eosio.wram.cpp

Impact Duplicated checks can cause extraneous execution costs.

Suggestion Remove the duplicated checks.

2.2 Note

2.2.1 Potential centralization risks

Description There are problems with the EOS WRAM contract that may increase the risk of centralization. Specifically, there is a mechanism called the **egress list** so that the accounts in this list is forbidden to transfer the WRAM token to other addresses. It effectively bans user from using the EOS WRAM contract. Besides, the corresponding RAM resource for the WRAM tokens of the banned users cannot sold or transferred.

```
// block transfers to any account in the egress list
```

```
void wram::check_disable_transfer( const name receiver )
{
    if (receiver == get_self()) { return; } // ignore self transfer (eosio.wram)

    egresslist _egresslist(get_self(), get_self().value);
    auto itr = _egresslist.find(receiver.value);
    check( itr == _egresslist.end(), "transfer disabled to account" );
}
```

Listing 2.3: src/egress.cpp

Feedback from the Project The egresslist is mostly to prevent users from sending to “eosio.ram” since this will result in loss of funds. Or any other system account that are not designed to receive `WRAM`.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS