BLOCKSEC

Security Audit Report for xRHEA Token

Date: July 31, 2026 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Rhea Finance
Target	xRHEA Token

Version History

Version	Date	Description
1.0	July 31, 2026	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2026-08-07 09:39:23+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 the Audit Target

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

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1] of xRHEA Token of Rhea Finance.

This incremental audit reviews the xRHEA staking contract upgrade, which introduces time locked unstake requests, configurable exit fees, fee distribution and retry mechanisms, and quota controlled instant unstaking for approved accounts. The audit further covers high precision virtual pricing, state migration, asynchronous NEP-141 callback handling, reward accounting, storage management, and the integration of the upgraded unstake workflow with the existing withdrawal implementation.

Note this audit only focuses on the smart contracts in the following directories/files:

- contracts/xrhea/src

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore 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. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
xRHEA Token	`Version 1`	`b6949f340febcda663156ebf34c578175917a320`
	`Version 2`	`76d33fa705774cc8ed8495a56374cfc47e285061`

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.

[1] https://github.com/ref-finance/xrhea-token

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 Target, 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 the Target.

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 solidity language), the underlying compiling toolchain 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 the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target 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 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 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 \ 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 five 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.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

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

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

Chapter 2 Findings

In total, we have **one** recommendation and **one** note.

- Recommendation: 1
- Note: 1

ID	Severity	Description	Category	Status
1	-	Reject storage unregistration with pending unstake requests	Recommendation	Fixed
2	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 Recommendation

2.1.1 Reject storage unregistration with pending unstake requests

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `storage_unregister()` removes a user's xRHEA fungible token account and refunds the associated storage deposit, but it does not verify whether the user has any pending `UnstakeRequest`. A user may invoke function `storage_unregister()` after function `request_unstake()`, leaving the unstake request recorded even though the corresponding xRHEA account has been removed. Consequently, function `get_account_unstake_requests()` may continue to return pending requests for an unregistered account, which can create inconsistent account state and ambiguity for frontends and external integrations.

```
23   #[payable]
24   #[pause(except(roles(Role::DAO)))]
25   fn storage_unregister(&mut self, force: Option<bool>) -> bool {
26       if force == Some(true) {
27           env::panic_str("Force unregister is unsupported");
28       }
29       if self.data().pending_legacy_unstake_counts.contains_key(&env::predecessor_account_id()) {
30           env::panic_str("cannot unregister while legacy unstake is pending");
31       }
32       if let Some((account_id, balance)) = self.data_mut().token.internal_storage_unregister(
             force) {
33           self.data_mut().current_account_num -= 1;
34           log!("Closed @{} with {}", account_id, balance);
35           true
36       } else {
37           false
38       }
39   }
```

Listing 2.1: contracts/xrhea/src/api/storage.rs

Suggestion Add check in function `storage_unregister()` to reject unregistration while the account still has any existing `UnstakeRequest`.

2.2 Note

2.2.1 Potential centralization risk

Introduced by `Version 1`

Description The `DAO` is authorized to perform several security sensitive operations, including upgrading the contract code, configuring exit fees and fee recipients, granting free unstake quotas, and modifying reward distribution parameters. These privileges are consistent with the protocol's governance model, but they also create a significant dependency on the security and integrity of the `DAO` governance process. If the `DAO`-controlled accounts or governance execution process are compromised or misused, an attacker could modify withdrawal conditions, redirect protocol fees, alter reward distribution, or deploy malicious contract logic, potentially disrupting the protocol or causing loss of user assets.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS