BLOCKSEC

Security Audit

Report for RHEA Token Ecosystem Contracts

Date: April 21, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	RHEA Finance
Target	RHEA Token Ecosystem Contracts

Version History

Version	Date	Description
1.0	April 21, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-05-05 14:23:34+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	Rust
Approach	Semi-automatic and manual verification

The target of this audit is the code repository of rhea-token[1], xrheatoken[2], orhea-token[3], orhea-vault[4] of RHEA Finance.

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
rhea-token	`Version 1`	`ad90818d7a34f7d5b50442a9ac0f9d2a865513d8`
	`Version 2`	`78f048e5583a729ed04bc98f8d7ca579d4838bc3`
xrhea-token	`Version 1`	`2ac8cbf7b05fd6d638955bf449b20b5dcd20c298`
	`Version 2`	`5d6c399697fd731e3ad7761be6f8ff7116044f91`
orhea-token	`Version 1`	`804c3b37cf36c6994e42271c49e6a277016afd6a`
	`Version 2`	`91ba499fae1cd0a418475e2892767d9e8fa77618`
orhea-vault	`Version 1`	`570618f91a27cd291f1dd6336ea9ac1c3a79131c`
	`Version 2`	`b39b3fc18a4266546891fafbf5af960c3f4b9ff1`

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 can-

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

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

[3] https://github.com/ref-finance/orhea-token

[4] https://github.com/ref-finance/orhea-vault

not 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 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 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 [5] and Common Weakness Enumeration [6]. 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 four categories:

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

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

[6] 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 find **three** potential issues. Besides, we also have **one** recommendation and **two** notes as follows:

- Low Risk: 3
- Recommendation: 1
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Lack of metadata validation in function `update_metadata()`	DeFi Security	Fixed
2	Low	Manipulable share price via account un-registration	DeFi Security	Fixed
3	Low	Unfair advantage for the first staker	DeFi Security	Fixed
4	-	Redundant entry in claim_history on roll-back failure	Recommendation	Fixed
5	-	Role-based permissions and upgrade-ability managed by DAO	Note	-
6	-	No interest accrued during the period from unstake to withdraw	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of metadata validation in function `update_metadata()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In both the `rhea-token` and `orhea-token` contracts, the `new()` function initializes the token and validates its `metadata` using function `assert_valid()` to ensure compliance with the `NEP-141` standard. However, in the `update_metadata()` function, which allows the contract owner to modify `metadata` post-deployment, this validation step is missing.

```
58  #[init]
59  pub fn new(owner_id: AccountId, total_supply: U128, metadata: FungibleTokenMetadata) -> Self {
60      require!(!env::state_exists(), "Already initialized");
61      metadata.assert_valid();
62      let mut this = Self {
63          owner_id: owner_id.clone(),
64          token: FungibleToken::new(StorageKey::FungibleToken),
65          metadata: LazyOption::new(StorageKey::Metadata, Some(&metadata)),
66      };
67      this.token.internal_register_account(&owner_id);
68      this.token.internal_deposit(&owner_id, total_supply.into());
69
70
71      near_contract_standards::fungible_token::events::FtMint {
72          owner_id: &owner_id,
73          amount: total_supply,
74          memo: Some("new tokens are minted"),
75      }
76      .emit();
77
78
79      this
80  }
81
82
83  #[payable]
84  pub fn update_metadata(&mut self, metadata: FungibleTokenMetadata) {
85      assert_one_yocto();
86      require!(self.owner_id == env::predecessor_account_id(), "Not allow");
87      let current_metadata = self.metadata.get().unwrap();
88      require!(
89          current_metadata.decimals == metadata.decimals,
90          "Can't change decimals"
91      );
92      self.metadata.set(&metadata);
93  }
```

Listing 2.1: rhea/src/lib.rs

```
58  #[init]
59  pub fn new(owner_id: AccountId, total_supply: U128, metadata: FungibleTokenMetadata) -> Self {
60      require!(!env::state_exists(), "Already initialized");
61      metadata.assert_valid();
62      let mut this = Self {
63          owner_id: owner_id.clone(),
64          token: FungibleToken::new(StorageKey::FungibleToken),
65          metadata: LazyOption::new(StorageKey::Metadata, Some(&metadata)),
66      };
67      this.token.internal_register_account(&owner_id);
68      this.token.internal_deposit(&owner_id, total_supply.into());
69
70
71      near_contract_standards::fungible_token::events::FtMint {
72          owner_id: &owner_id,
73          amount: total_supply,
74          memo: Some("new tokens are minted"),
75      }
76      .emit();
77
78
79      this
80  }
81
82
83  #[payable]
84  pub fn update_metadata(&mut self, metadata: FungibleTokenMetadata) {
85      assert_one_yocto();
86      require!(self.owner_id == env::predecessor_account_id(), "Not allow");
87      let current_metadata = self.metadata.get().unwrap();
88      require!(
89          current_metadata.decimals == metadata.decimals,
90          "Can't change decimals"
91      );
92      self.metadata.set(&metadata);
93  }
```

Listing 2.2: orhea/src/lib.rs

Impact The token `metadata` may not comply with the `NEP-141` standard.

Suggestion Add check to ensure that the provided token `metadata` is valid.

2.1.2 Manipulable share price via account unregistration

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The xRHEA token functions as a vault and minted to users accordingly when they stake RHEA tokens. However, the implementation of function storage_unregister() allows users to forcibly unregister their accounts. This process directly removes the user's internal account and burns their xRHEA token balance.

Since the share price of xRHEA is calculated as the total staked RHEA divided by the total supply of xRHEA tokens, forcibly burning xRHEA tokens reduces the total supply of xRHEA tokens, causing an artificial spike in the token price. Malicious users could exploit this mechanism to manipulate xRHEA's share price and potentially conduct price-based attacks on external protocols relying on this price.

```
36  #[payable]
37  #[pause(except(roles(Role::DAO)))]
38  fn storage_unregister(&mut self, force: Option<bool>) -> bool {
39      #[allow(unused_variables)]
40      if let Some((account_id, balance)) =
41          self.data_mut().token.internal_storage_unregister(force)
42      {
43          self.data_mut().current_account_num -= 1;
44          log!("Closed @{} with {}", account_id, balance);
45          true
46      } else {
47          false
48      }
49  }
```

Listing 2.3: xrhea/src/api/storage.rs

```
 8  pub fn internal_storage_unregister(
 9      &mut self,
10      force: Option<bool>,
11  ) -> Option<(AccountId, Balance)> {
12      assert_one_yocto();
13      let account_id = env::predecessor_account_id();
14      let force = force.unwrap_or(false);
15      if let Some(balance) = self.accounts.get(&account_id) {
16          if balance == 0 || force {
17              self.accounts.remove(&account_id);
18              self.total_supply -= balance;
19              Promise::new(account_id.clone()).transfer(
20                  self.storage_balance_bounds().min.saturating_add(NearToken::from_yoctonear(1)),
21              );
22              Some((account_id, balance))
23          } else {
24              env::panic_str(
25                  "Can't unregister the account with the positive balance without force",
26              )
27          }
28      } else {
29          log!("The account {} is not registered", &account_id);
30          None
31      }
32  }
```

Listing 2.4: near-contract-standards/src/fungible_token/storage_impl.rs

Impact `xRHEA`'s share price can be manipulated.

Suggestion Restrict or redesign function `storage_unregister()` to prevent unintended burning of `xRHEA` tokens.

2.1.3 Unfair advantage for the first staker

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `xrhea-token` contract, rewards can be added before any user has staked, i.e., when the total supply of `xRHEA` is still zero. Since the `distribute_reward()` function updates the total staked amount (`locked_token_amount`) without checking whether any `xRHEA` has been minted, this increases the underlying asset pool without issuing corresponding shares. As a result, the first staker receives `xRHEA` at a 1:1 ratio with their stake, but can immediately unstake to capture all previously added rewards. This creates an unfair advantage for early stakers.

```
13  pub fn ft_on_transfer(
14      &mut self,
15      sender_id: AccountId,
16      amount: U128,
17      msg: String,
18  ) -> ContractResult<PromiseOrValue<U128>> {
19      self.distribute_reward();
20      let transfer_amount: u128 = amount.into();
21
22
23      let token_id = env::predecessor_account_id();
24      if token_id != self.data().locked_token_id {
25          return Err(FtError::InvalidTokenId(token_id).into());
26      }
27
28
29      let message =
30          serde_json::from_str::<TokenReceiverMessage>(&msg).map_err(GlobalError::InvalidJson)?;
31
32
33      match message {
34          TokenReceiverMessage::Stake => {
35              self.internal_stake(&sender_id, transfer_amount)?;
36              Ok(PromiseOrValue::Value(U128(0)))
37          }
38          TokenReceiverMessage::Reward => {
39              self.internal_add_reward(&sender_id, transfer_amount);
40              Ok(PromiseOrValue::Value(U128(0)))
41          }
42      }
```

```
43  }
```

Listing 2.5: xrhea/src/ft_transfer.rs

```
17  pub fn unstake(&mut self, amount: U128) -> ContractResult<Promise> {
18      assert_one_yocto();
19
20
21      // Checkpoint
22      self.distribute_reward();
23
24
25      let account_id = env::predecessor_account_id();
26      let burned_amount: u128 = amount.into();
27
28
29      if self.data().token.total_supply == 0 {
30          return Err(FtError::TotalSupplyEmpty.into());
31      }
32
33
34      let unlocked_amount = u128_ratio(
35          burned_amount,
36          self.data().locked_token_amount,
37          self.data().token.total_supply,
38      );
39
40
41      self.data_mut()
42          .token
43          .internal_withdraw(&account_id, burned_amount);
44
45
46      let metadata = self.data().metadata.get().as_ref().unwrap();
47
48
49      if self.data().token.total_supply < 10u128.pow(metadata.decimals as u32) {
50          return Err(FtError::KeepAtLeastOneXrhea.into());
51      }
52
53
54      self.data_mut().locked_token_amount -= unlocked_amount;
55
56
57      Ok(ext_ft_core::ext(self.data().locked_token_id.clone())
58          .with_attached_deposit(NearToken::from_yoctonear(1))
59          .with_static_gas(GAS_FOR_TOKEN_TRANSFER)
60          .ft_transfer(account_id.clone(), unlocked_amount.into(), None)
61          .then(
62              Self::ext(env::current_account_id())
63                  .with_static_gas(GAS_FOR_AFTER_TOKEN_TRANSFER)
64                  .callback_post_unstake(
65                      account_id.clone(),
```

```
66                      unlocked_amount.into(),
67                      burned_amount.into(),
68                  ),
69          ))
70  }
```

Listing 2.6: xrhea/src/unstake.rs

```
78  pub fn distribute_reward(&mut self) {
79      let cur_time = ms_to_sec(current_timestamp_ms());
80      let new_reward = self.try_distribute_reward(cur_time);
81      if new_reward > 0 {
82          self.data_mut().undistributed_reward_amount -= new_reward;
83          self.data_mut().locked_token_amount += new_reward;
84      }
85      self.data_mut().prev_distribution_time_in_sec =
86          std::cmp::max(cur_time, self.data().reward_genesis_time_in_sec);
87  }
```

Listing 2.7: xrhea/src/xrhea.rs

Impact The first staker may receive disproportionate rewards.

Suggestion Disallow reward deposits when total share supply is zero.

2.2 Additional Recommendation

2.2.1 Redundant entry in claim_history on rollback failure

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `claim()` function lets designated users receive airdropped `ORHEA` tokens, recording their claim amount in an `IterableMap` named `claim_history`. Since the token transfer is a cross-contract asynchronous call, if it fails, a rollback occurs in the next block. The current rollback logic checks whether the user exists in `claim_history` to decide between subtraction or insertion. However, a user is always present in the map once recorded, with only the value possibly being zero. As a result, entries with a value of zero can persist unnecessarily. Instead of checking for presence, the logic should check if the resulting amount after subtraction is zero and remove the entry accordingly.

```
90  #[handle_result(aliased)]
91  #[payable]
92  #[pause(except(roles(Role::DAO)))]
93  pub fn claim(&mut self) -> ContractResult<Promise> {
94      assert_one_yocto();
95      let receiver_id = env::predecessor_account_id();
96      match self.data_mut().airdrop.remove(&receiver_id) {
97          Some(claim_amount) => {
98              self.data_mut()
99                  .claim_history
100                 .entry(receiver_id.clone())
```

```
101                 .and_modify(|v| *v += claim_amount)
102                 .or_insert(claim_amount);
103             Ok(self.transfer_airdrop_token(&receiver_id, claim_amount))
104         }
105         None => Err(AirdropError::NoAirdrop(receiver_id.clone()).into()),
106     }
107 }
```

Listing 2.8: orhea_vault/src/lib.rs

```
24  #[private]
25  pub fn transfer_airdrop_token_callback(
26      &mut self,
27      receiver_id: AccountId,
28      amount: U128,
29  ) -> bool {
30      let promise_success = is_promise_success();
31      if !promise_success {
32          self.data_mut()
33              .airdrop
34              .entry(receiver_id.clone())
35              .and_modify(|v| *v += amount.0)
36              .or_insert(amount.0);
37          self.data_mut()
38              .claim_history
39              .entry(receiver_id.clone())
40              .and_modify(|v| *v -= amount.0)
41              .or_insert(amount.0);
42      }
43      Event::ClaimAirdrop {
44          account_id: &receiver_id,
45          amount,
46          success: promise_success,
47      }
48      .emit();
49      promise_success
50  }
```

Listing 2.9: orhea_vault/src/token_transfer.rs

Suggestion Remove `claim_history` entry when the updated value becomes zero.

Feedback from the project The `claim_history` should be stored for each user, and the only case where it would be 0 is if the first claim attempt fails, so there is no need to explicitly clear it. The use of `or_insert` is unnecessary and has been removed.

2.3 Note

2.3.1 Role-based permissions and upgradeability managed by DAO

Introduced by `Version 1`

Description In the current implementation, key protocol functions—such as pausing, parameter configuration, and contract upgrades—are controlled via `DAO` governance. This mitigates traditional centralization concerns, as privileged actions are no longer executed by individual accounts but are subject to `DAO` decisions. However, the effectiveness of this decentralization ultimately depends on the security and transparency of the `DAO` itself, including its voting process and key management.

2.3.2 No staking rewards during initial 30-day period

Introduced by `Version 1`

Description The contract permits users to stake tokens immediately after deployment. However, actual reward accrual does not begin until 30 days later, as determined by `reward_genesis_time_in_sec`. During this initial phase, users who stake early will not receive any rewards, regardless of how long their tokens remain staked.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS