BLOCKSEC

Security Audit Report for multichain-account

Date: November 14, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Rhea Finance
Target	multichain-account

Version History

Version	Date	Description
1.0	November 14, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-11-14 05:47:13+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 [1] of multichain-account of Rhea Finance.

The multichain account system is a NEAR blockchain project designed to unify user wallets across multiple blockchains. Each user is provisioned with a dedicated account contract through the account manager, which consolidates the user's public keys from different chains. Using this account mechanism, users can sign business messages with wallets from any supported chain, enabling direct interactions with decentralized applications and asset transfers on the NEAR network. The system is designed to support seamless multi-chain operations while maintaining strong security and flexibility, providing a reliable foundation for cross-chain decentralized applications.

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

- contracts/account_manager/src/*
- contracts/multichain_account/src/*
- contracts/common/src/*

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope 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
multichain-account	Version 1	103ca3cba5c708486650e91010e22f308f288535
	Version 2	cdbe27023314fb22d65861707e942669686ce3f9

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.

[1] https://github.com/ref-finance/multichain-account

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 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 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 found **four** potential security issues. Besides, we have **one** recommendation and **three** notes.

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

ID	Severity	Description	Category	Status
1	High	Insufficient check on the `wallets` while creating `multichain_account`	Security Issue	Fixed
2	High	Lack of predecessor validation in function `ft_on_transfer()`	Security Issue	Fixed
3	Medium	Potential fund locked due to lack of checks for `wallets`	Security Issue	Fixed
4	Low	Incorrect registration logic in function `create_mca_callback()`	Security Issue	Fixed
5	-	Add a limit to the number of `register_dapps` and `tx_requests`	Recommendation	Confirmed
6	-	Potential centralization risks	Note	-
7	-	Ensure the `near_value` of each token is valid and accurate	Note	-
8	-	Relayer should validate token value and amount when processing `GasPayment`	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Insufficient check on the `wallets` while creating `multichain_account`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `account_manager`, the function `ft_on_transfer()` allows users to create a `multichain_account` containing multiple wallets from different chains, where each wallet must be uniquely bound to one `mca_id`. However, the validation mechanism is insufficient because it permits malicious users to front-run transactions. Specifically, an attacker can create a legal `multichain_account` including the victim's wallet, causing the victim's transaction to fail while the account is created. As a result, if the victim participates in DeFi protocols with this compromised account, the attacker can seize their assets through the injected wallet.

```
    let prepaid_gas = env::prepaid_gas();
    must!(
            prepaid_gas >= MIN_GAS_FOR_FT_ON_TRANSFER,
            GlobalError::InsufficientGas {
                required: MIN_GAS_FOR_FT_ON_TRANSFER.as_tgas(),
                provided: prepaid_gas.as_tgas()
            }
        );
        let token_id = env::predecessor_account_id();
        let amount = amount.0;
        let intents_message =
            serde_json::from_str::<IntentsMessage>(&msg).map_err(GlobalError::InvalidJson)?;

        must!(
            intents_message.wallets.len() <= self.max_wallets_per_mca as usize,
            WalletError::ExceedLimit(self.max_wallets_per_mca)
        );
        let mut wallets_set = HashSet::new();
        for wallet in &intents_message.wallets {
            wallet.assert_valid();
            if let Some(mca_id) = self.wallet_to_mca_id.get(wallet) {
                return Err(WalletError::AlreadyBound(mca_id.clone()).into());
            }
            must!(
                wallets_set.insert(wallet.clone()),
                WalletError::DuplicateWallet(wallet.clone())
            );
        }
```

Listing 2.1: contracts/account_manager/src/interfaces/token_receiver.rs

Impact This vulnerability allows attackers to seize user assets by front-running `multichain_account` creation.

Suggestion Ensure the wallets used in the account creation are trustworthy.

2.1.2 Lack of predecessor validation in function `ft_on_transfer()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `multichain_account`, the function `ft_on_transfer()` handles messages triggered by token transfers. However, the `predecessor_account_id` is not validated by the function, allowing arbitrary accounts to invoke it directly instead of being restricted to legitimate `FT` contracts.

As a result, a malicious user could craft a direct call to function `ft_on_transfer()` with a spoofed `sender_id` (e.g., setting it to `ROOT_ACCOUNT_ID`) and register fake `dApps`. Since the function proceeds to invoke function `process_msg()` without verifying the caller's authenticity, the contract could mistakenly attach storage deposits, which leads to loss of funds.

```
    pub fn ft_on_transfer(
        &mut self,
        sender_id: AccountId,
        amount: U128,
        msg: String,
    ) -> PromiseOrValue<U128> {
        Self::ext(env::current_account_id()).process_msg(
            sender_id,
            env::predecessor_account_id(),
            amount,
            msg,
        );
        PromiseOrValue::Value(U128(0))
    }
```

Listing 2.2: contracts/multichain_account/src/lib.rs

Impact This flaw enables unauthorized storage deposits, leading to loss of funds.

Suggestion Restrict access to registrations by validating `env::predecessor_account_id()` against the whitelist `FT` contracts.

2.1.3 Potential fund locked due to lack of checks for `wallets`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `account_manager`, the function `ft_on_transfer()` allows users to create a multichain account and transfer their tokens after deducting the required fees. However, the function does not validate whether the wallet list is empty. If an empty list is provided, the account is still created and the transferred funds are sent to it, making them permanently inaccessible since no valid wallet can operate the account.

```
        must!(
            intents_message.wallets.len() <= self.max_wallets_per_mca as usize,
            WalletError::ExceedLimit(self.max_wallets_per_mca)
        );
        let mut wallets_set = HashSet::new();
        for wallet in &intents_message.wallets {
            wallet.assert_valid();
            if let Some(mca_id) = self.wallet_to_mca_id.get(wallet) {
                return Err(WalletError::AlreadyBound(mca_id.clone()).into());
            }
            must!(
                wallets_set.insert(wallet.clone()),
                WalletError::DuplicateWallet(wallet.clone())
            );
        }
```

Listing 2.3: contracts/account_manager/src/interfaces/token_receiver.rs

Impact User funds may become permanently locked because the created account has no valid wallet bound to it.

Suggestion Revise the logic to ensure that `wallets` are not empty before creating the `mca`.

2.1.4 Incorrect registration logic in function `create_mca_callback()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `account_manager`, the function `ft_on_transfer()` is responsible for handling token transfers and initiating the creation of a new multichain account through an asynchronous call. Once the multichain account is successfully created, `create_mca_callback()` function continues the process by transferring tokens to the new account through the function `notify_mca()` and triggering the function `ft_on_transfer()` within the newly created account to complete the registration process with the specified decentralized application.

However, the current implementation contains an incorrect logic condition. The function `notify_mca()` will not be invoked if the remaining amount is zero. As a result, even though the protocol collects the required registration fee, the user's account registration is not finalized.

```
            if remain_amount.0 > 0 {
                self.notify_mca(token_id, remain_amount.0, mca_id, msg)
            } else {
                PromiseOrValue::Value(U128(0))
            }
```

Listing 2.4: contracts/account_manager/src/interfaces/token_receiver.rs

Impact The registration process may not be completed.

Suggestion Revise the logic accordingly.

2.2 Recommendation

2.2.1 Add a limit to the number of `register_dapps` and `tx_requests`

Status Confirmed

Introduced by `Version 1`

Description When creating a `multichain_account`, users can choose to register `register_dapps`. However, without limiting the number of registrations, the gas cost may exceed the transaction gas limit. Similarly, in function `internal_process_business_with_signature()`, the number of `tx_requests` to be executed has no cap, which can also lead to out of gas.

```
    let (dapp_registration_near_cost, dapp_registration_token_cost) = if let Some(ref dapps) =
```

Listing 2.5: contracts/account_manager/src/interfaces/token_receiver.rs

```
process_tx_requests(business.tx_requests)
```

Listing 2.6: contracts/multichain_account/src/lib.rs

Suggestion Introduce maximum limits for both the number of dapp registrations and the number of transaction requests to ensure that transaction gas costs remain within safe bounds.

Feedback from the project It's designed this way to provide maximum flexibility. Transactions and gas estimation are handled off-chain, and the contract is responsible for executing these transactions correctly, which also ensures they fail if gas runs out.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In this protocol, privileged roles like the `DAO` can conduct sensitive operations (e.g., updating `multichain_account` contract code and fee collection). Besides, the account that holds the full access key of the contract `account_mananger` can upgrade the contract. If the private key of this account is lost or maliciously exploited, it could pose a significant risk to the protocol, leading to user asset loss.

2.3.2 Ensure the `near_value` of each token is valid and accurate

Introduced by `Version 1`

Description In the function `ft_on_transfer()`, when processing dapp registration intents, the contract calculates the total registration fee based on the quoted `NEAR` token creation fee and the stored `near_value` for each token. Since different tokens may have different decimal precisions, it is essential to ensure that the `near_value` is properly scaled according to each token's decimal configuration. Maintaining updated and correctly scaled `near_value` records is critical to prevent potential financial discrepancies or miscalculations in fee collection.

2.3.3 Relayer should validate token value and amount when processing `GasPayment`

Introduced by `Version 1`

Description In the contract `multichain_account`, transaction submission relies on relayers to broadcast users' signed transactions. Users may include a `GasPayment` operation in their transaction requests to pay service fees to the relayer using either `FT` tokens or `NEAR`.

When processing a `GasPayment` request, the contract executes an `ft_transfer()` call to the specified token without verifying the token's value or the amount. If the token has no value, or the amount is unreasonable, the relayer may incur unnecessary gas costs or experience failed transfers. Repeatedly processing such transactions could gradually drain the relayer's funds. Relayers should check that the token used for `GasPayment` has sufficient value and that the specified amount is reasonable before executing the transfer.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS