# Security Audit Report for RWA Contracts

**Date:** October 17, 2024 **Version:** 1.1

**Contact:** contact@blocksec.com

# Contents

## Report Manifest

| Item | Description |
|---|---|
| Client | Matrixdock |
| Target | RWA Contracts |

## Version History

| Version | Date | Description |
|---|---|---|
| 1.0 | September 11, 2024 | First release |
| 1.1 | October 17, 2024 | Update commit hash |

## Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-10-17 12:23:45+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 | Solidity |
| Approach | Semi-automatic and manual verification |

The focus of this audit is on the RWA Contracts of Matrixdock [1]. RWA Contracts issue `MToken`s, representing real-world assets. Users can obtain `BullionBFT` by packing `MToken`s and redeem `BullionNFT` for `MToken`s by unpacking. Additionally, RWA Contracts enable users to transfer, redeem, and bridge assets to other supported chains, as well as perform other operations on `BullionNFT`s and `MToken`s.

Please note that the audit scope is limited to the smart contracts in the `contracts` folder in the repository. The contracts for interface and testing purposes are not within the scope of this 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.

| Project | Version | Commit Hash |
|---|---|---|
| RWA Contracts | `Version 1` | `51e74fbfc1f4d683bff9272d13d1473e1015bdb0` |
| | `Version 2` | `8cfe33961d7b00329fea7a9746031bdafa34f99f` [2] |

## 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

[1] https://github.com/Matrixdock-RWA/RWA-Contracts/

[2] In Version 1.1 of this audit report, the `BullionMinter` contract was added to the repository. After reviewing the file, we found no security issues. The commit hash for `Version 2` has been updated accordingly.

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 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

| 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:

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

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

- **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.
- **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 **five** recommendations and **two** notes.

- Medium Risk: 1
- Low Risk: 1
- Recommendation: 5
- Note: 2

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | Medium | Lack of blacklist checks in the `unpack` function | DeFi Security | Fixed |
| 2 | Low | Lack of checks on oracle validity | DeFi Security | Fixed |
| 3 | - | Add sanity checks in the `MToken` contract | Recommendation | Acknowledged |
| 4 | - | Refactor the method of refunding fees | Recommendation | Fixed |
| 5 | - | Add validity checks on functions setting key parameters | Recommendation | Fixed |
| 6 | - | Emit the correct event for cross chain operations | Recommendation | Fixed |
| 7 | - | Emit the `AllowedPeer` event | Recommendation | Fixed |
| 8 | - | Reuse of the IDs in the `BullionNFT` contract | Note | - |
| 9 | - | Potential centralization risks | Note | - |

The details are provided in the following sections.

## 2.1 DeFi Security

### 2.1.1 Lack of blacklist checks in the `unpack` function

**Severity** Medium

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `unpack` function of the `BullionNFT` contract allows the NFT owners to redeem `BullionNFT`s for the corresponding amount of `MToken`s. However, this function does not verify if the caller is blocked, potentially enabling a blocked user to invoke the function and receive `MToken`s.

```
function unpack(uint bullion) public {
    uint amount = _getAmount(bullion);
    if (ownerOf(bullion) != msg.sender) {
        revert NotNftOwner(bullion, msg.sender);
    }
    delete packedCoins[bullion];
    IMToken(mtokenContract).unpack(msg.sender, amount);
    _burn(bullion);
}
```

**Listing 2.1:** contracts/BullionNFT.sol

**Impact** Blocked users can invoke the `unpack` function to receive `MToken`s.

**Suggestion** Add blacklist checks for the caller (i.e., `msg.sender`).

### 2.1.2 Lack of checks on oracle validity

**Severity** Low

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `MToken` contract relies on Chainlink to fetch the proof of reserve (PoR) data, as shown in the `increaseMintBudget` function in the following code segment. However, this function does not verify that the last updated timestamp of the returned reserve data is within a reasonable time window.

```
function increaseMintBudget(uint112 mintBudgetDelta) public onlyOperator {
    uint _usedReserve = usedReserve + mintBudgetDelta;
    // prettier-ignore
    (
        /*uint80 roundID*/,
         int reserveFromFeed ,
        /*uint startedAt*/ ,
        /*uint timestamp*/,
        /*uint80 answeredInRound*/
    ) = AggregatorV3Interface(reserveFeed).latestRoundData();
    if (int(_usedReserve) > reserveFromFeed) {
        revert ReserveNotEnough(reserveFromFeed, int(_usedReserve));
    }
    mintBudget += uint112(mintBudgetDelta);
    usedReserve = uint112(_usedReserve);
}
```

**Listing 2.2:** contracts/MTokenMain.sol

**Impact** The protocol may use outdated reserve data due to insufficient validation checks.

**Suggestion** Verify the validity of the reserve data.

## 2.2 Additional Recommendation

### 2.2.1 Add sanity checks in the `MToken` contract

**Status** Acknowledged

**Introduced by** `Version 1`

**Description** In the `MToken` contract, the `ccSendToken` function allows users to make cross-chain transfers. It performs a `_checkBlocked` check on both the sender and the receiver by invoking the `msgOfCcSendToken` function.

```
function msgOfCcSendToken(
    address sender,
    address receiver,
    uint256 value
) public view returns (bytes memory message) {
    _checkBlocked(sender);
    _checkBlocked(receiver);
    return abi.encode(TagSendToken, abi.encode(sender, receiver, value));
}

// called by the messager contract to initialize a cross-chain token transfer
function ccSendToken(
    address sender,
    address receiver,
    uint256 value
) public onlyMessager returns (bytes memory message) {
    if (disableCcSend) {
        revert CcSendDisabled();
    }
    _checkZeroValue(value);
    _burn(sender, value);
    emit CCSendToken(sender, receiver, value);
    return msgOfCcSendToken(sender, receiver, value);
}
```

**Listing 2.3:** contracts/MToken.sol

The `transfer` and `transferFrom` functions allow users to transfer tokens to other addresses. However, unlike the `ccSendToken` function, there functions do not check if the recipient address is blocked. This omission could lead to unexpected behavior, such as a non-blocked address transferring tokens to a blocked address.

```
function transfer(
    address _recipient,
    uint256 _amount
) public virtual override onlyNotBlocked returns (bool) {
    if (_recipient == address(this)) {
        revert TransferToContract();
    }
    return super.transfer(_recipient, _amount);
}

function transferFrom(
    address _sender,
    address _recipient,
    uint256 _amount
) public virtual override onlyNotBlocked returns (bool) {
    if (_recipient == address(this)) {
        revert TransferToContract();
    }
    _checkBlocked(_sender);
    return super.transferFrom(_sender, _recipient, _amount);
}
```

**Listing 2.4:** contracts/MToken.sol

**Suggestion** Implement proper checks to ensure the recipient address is not blocked.

**Feedback from the Project** The permission of the blocked users are restricted to only receiving, not sending tokens. The check for receivers in `ccSendToken` is to ensure that the blocked users are not able to send on the destination chain when the blacklist is not timely synchronized across chains.

### 2.2.2 Refactor the method of refunding fees

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `sendDataToChain` function uses `send` to transfer native token to the `msg.sender` for fee refunds. However, both `send` and `transfer` impose strict gas limits (i.e., 2300 gas), which may not be sufficient for all scenarios. It is recommended to use `call` or the `sendValue` function from OpenZeppelin's `Address` library instead.

```
function sendDataToChain(
    uint64 destinationChainSelector,
    address messageReceiver,
    bytes calldata extraArgs,
    bytes memory data
) internal returns (bytes32 messageId) {
    Client.EVM2AnyMessage memory evm2AnyMessage = Client.EVM2AnyMessage({
        receiver: abi.encode(messageReceiver),
        data: data,
        tokenAmounts: new Client.EVMTokenAmount[](0),
        extraArgs: extraArgs,
        feeToken: address(0)
    });
    uint256 fee = IRouterClient(getRouter()).getFee(
        destinationChainSelector,
        evm2AnyMessage
    );
    if (msg.value < fee) {
        revert InsufficientFee(fee, msg.value);
    }
    messageId = IRouterClient(getRouter()).ccipSend{value: fee}(
        destinationChainSelector,
        evm2AnyMessage
    );
    if (msg.value - fee > 0) {
        bool success = payable(msg.sender).send(msg.value - fee);
        require(success, "MTokenMessager: TRANSFER_FAILED");
    }
    emit CCSendToken(messageId, data);
    return messageId;
}
```

Listing 2.5: contracts/MTokenMessager.sol

**Suggestion** Use `call` or OpenZeppelin's `sendValue` instead of `send` for refunding fees.

### 2.2.3 Add validity checks on functions setting key parameters

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the `MToken`, `MTokenMessager`, and `BullionNFT` contracts, some functions that set key parameters do not validate their input values:

- In the `MToken` contract, the `setDelay`, `setMessager`, `setNFTContract`, `setRevoker`, and `setOperator` functions do not properly check the validity of input values.
- In the `MTokenMessager` and `BullionNFT` contracts, the `setAllowedPeer` and `setPackSigner` functions lack proper validation.

For address parameters, it is recommended to check whether the new address is zero. For the `delay` parameter, it is recommended to establish a minimum threshold in the `MToken` contract to ensure the validity of the delay mechanism.

**Suggestion** Add proper validity checks for input values of these functions.

### 2.2.4 Emit the correct event for cross chain operations

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the `MTokenMessager` contract, the `CCSendMintBudget` event is defined but not emitted, and the emission of the `CCSendToken` event is improperly placed in the `sendDataToChain` function. The absence of the `CCSendMintBudget` event and the misplacement of the `CCSendToken` event can result in incorrect event emissions when the `sendMintBudgetToChain` function is invoked.

**Suggestion** Refactor the relevant functions to ensure correct event emissions.

### 2.2.5 Emit the `AllowedPeer` event

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `AllowedPeer` event is defined but not used in the `setAllowedPeer` function of the `MTokenMessager` contract.

**Suggestion** Emit the `AllowedPeer` event.

## 2.3 Note

### 2.3.1 Reuse of the IDs in the `BullionNFT` contract

**Introduced by** `Version 1`

**Description** The NFT ID (i.e., `bullion`) becomes reusable in the `BullionNFT` contract after the corresponding NFT is burned.

**Feedback from the Project** This behavior is by design. Each NFT ID in the `BullionNFT` contract uniquely identifies a `bullion` as a real-world asset. Packing mints an NFT to represent that a user (i.e., the NFT receiver) has obtained a `bullion`. Once the user unpacks an NFT, the corresponding `bullion` no longer belongs to that user. Consequently, the NFT ID can be reused, allowing the NFT to be minted again and obtained by other users who are packing with that ID.

### 2.3.2 Potential centralization risks

**Introduced by** `Version 1`

**Description** The operator has the privilege to mint and redeem tokens in the `MToken` contract, as well as to pack and unpack `MToken`s in the `BullionNFT` contract. Additionally, the admin can set sensitive configurations, such as the operator address. Invoking privileged operations or altering key parameters can significantly affect the functionality of the contracts, potentially rendering them unusable or placing them in an incorrect state.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS