SECURITY AUDIT REPORT

for

KYOKO-CCAL

Prepared By: yeye

February 25, 2022

Document Properties

Client	Kyoko-CCAL
Title	Security Audit Report
Target	Ethereum Smart Contract
Version	1.0
Author	yeye
Auditors	Darker, Haoqi,Ye ye
Reviewed by	Haoqi
Approved by	yeye
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 25, 2022	yeye	Perview Release
final	Mar 1, 2022	Haoqi	final Report

Contact

For more information about this document and its contents, please contact MatrixSec Inc.

Name	yeye
Phone	+65 3028 2468
Email	yeye@matrixsec.org

Contents

1 Introduction Kyoko-CCAL

The Rings were not forged as weapons of war or conquest, that is not their power.
Those who made the Ring did not desire to power, to rule, to gather wealth, but to understand, to make, to heal, to preserve all things untainted.

This document outlines our audit results.

1.1 About Ethereum Smart Contract

The **Ethereum Smart Contract** is simply a program that runs on the Ethereum blockchain. It's a collection of code (its functions) and data (its state) that resides at a specific address on the **Ethereum blockchain**.

Smart contracts are a type of **Ethereum** account. This means they have a balance and they can send transactions over the network. However they're not controlled by a user, instead they are deployed to the network and run as programmed. User accounts can then interact with a smart contract by submitting transactions that execute a function defined on the smart contract. Smart contracts can define rules, like a regular contract, and automatically enforce them via the code. Smart contracts cannot be deleted by default, and interactions with them are irreversible.

1.2 About MatrixSec

MatrixSec Inc. is a top international security company t was founded in September 2019.
Founded by a team with more than 10 years of experience in frontline network security attack and defense.
MatrixSec is committed to providing customers with mature and reliable security solutions with the concept of **"Secure Before You Feel Ready"**.

1.3 About Kyoko.finance

Kyoko.finance is a cross-chain GameFi NFT lending market for guilds and players.

Kyoko.finance-CCAL means Cross-chain asset lending, that is, cross-chain asset lending. Users rent out all assets (ERC721) on the front-end page. Game assets come from various chains. The corresponding assets selected by the user are frozen on the deployment (main chain) and ERC20 is returned after use. Can leave the country.

The basic information of Ethereum Smart Contract is as follows:

Table 1.1: Basic Information of Ethereum Smart Contract

Item	Description
Issuer	Kyoko-CCAL
Website	https://www.kyoko.finance/
Type	Ethereum Smart Contract
Platform	solidity
Audit Method	Whitebox
Latest Audit Report	February 25, 2022

Table 1.2: Vulnerability Severity Classification

Impact		Likelihood: *High*	Likelihood: *Medium*	Likelihood: *Low*
Impact	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

Likelihood

1.4 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [4]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Parameters Injection
	Money-Giving Bug
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Random Number Security
	Delegate Call Security
	Digital Asset Escrow
	Contract Update Security
	Reordering Attack
	False Top-up
	Timestamp Dependency Attack

To evaluate the risk, we go through a vulnerability checklist and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private test environment and run tests to confirm the findings. If necessary, we would additionally develop a Proof-of-Concept to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [3], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.5 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Kyoko-CCAL implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	3	■ ■ ■
Informational	1	■
Total	4	■ ■ ■ ■

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 0 critical-severity vulnerability, 0 high-severity vulnerabilities, 0 medium-severity vulnerabilities, 0 low-severity vulnerabilities, and 4 informational recommendations.

Table 2.1: Key Ethereum Smart Contract Audit Findings

ID	Severity	Title	Category	Status
MSVE-001	Low	SPOF in releaseTokenEmergency withdraw methods and CreditSystem	Coding Practices	Informed
MSVE-002	Info	Liquidate liquidation logic should also be added to repayAsset	Coding Practices	Ignored
MSVE-003	Low	A tag with the chainId to assets is recommended to be added	Coding Practices	Ignored
MSVE-004	Low	Risks of whenNotPaused	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 SPOF in releaseTokenEmergency withdraw methods and CreditSystem

- ID: MSVE-001
- Severity: Low
- Likelihood: N/A
- Impact: N/A
- Target: `~None`
- Category: Coding Practices [2]
- CWE subcategory: CWE-1041 [1]

Description

The **releaseTokenEmergency** and **withdraw** methods belong to the backdoor that can be manipulated by the administrator at a single point.
CreditSystem belongs to the unsecured credit whitelist, and administrators should pay attention to the single point of risk.
Cross-chain Bot addresses should pay attention to private key security.

```
    function releaseTokenEmergency(
        address game,
        address user,
        uint internalId
    ) public onlyManager {
        bytes memory key = getFreezeKey(game, internalId);
```

Listing 3.1: KyokoCCAL.sol

```
    function withdraw(uint256 amount) public onlyManager {
        uint balance = IERC20Upgradeable(allowCurrency).balanceOf(address(this));
        require(amount <= balance, "no enough balance");
        SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(allowCurrency), payable(
            vault), amount);

        emit WithDraw(vault, amount);
    }
```

Listing 3.2: KyokoCCAL.sol

Kyoko-CCAL Reply:

After releaseTokenEmergency and withdraw are officially launched, they will be handed over to multi-signature management.
CreditSystem will also be handed over to multi-signature management.

3.2 Liquidate liquidation logic should also be added to repayAsset

- ID: MSVE-002
- Severity: Low
- Likelihood: N/A
- Impact: N/A
- Target: ~None
- Category: Coding Practices [2]
- CWE subcategory: CWE-1041 [1]

Description

The liquidate liquidation logic should also be added to **repayAsset**.
otherwise there may be two situations:

1. It is overdue but the interest has not reached the **totalAmount**. At this time, the borrower repays the NFT and only pays the interest part (the borrower defaults but does not take responsibility, and the lender loses).

2. Regardless of whether it is overdue or not, the interest reaches the **totalAmount**, the borrower repays the **NFT**, and the system also deducts the full **totalAmount**. (At this time, the NFT and Token are gone, resulting in the loss of the borrower)

```
    function repayAsset(address game, address holder, uint internalId) public
        whenNotPaused {

        ValidateLogic.checkRepayAssetPara(game, _msgSender(), internalId, nftMap);

        DepositTool storage asset = nftMap[internalId];

        for (uint idx; idx < asset.toolIds.length; idx++) {
            IERC721Upgradeable(game).safeTransferFrom(_msgSender(), address(this), asset
                .toolIds[idx]);
        }

        _afterRepay(game, holder, _msgSender(), internalId);
    }
```

Listing 3.3: KyokoCCAL.sol

Kyoko-CCAL Reply:

1. By default, the lender will start liquidation only after the asset has been retrieved.
2. We will guide users to set a **totalAmount** that is far more than the value of **NFT** itself.

As for the situation where **NFT** and Token are gone, it will only be generated when the interest set by the lender is very high. And this is also known when the other party lends the **NFT**, so it is not considered for the time being.

3.3 A tag with the chainId to assets is recommended to be added

- ID: MSVE-003
- Severity: info
- Likelihood: N/A
- Impact: N/A
- Target: ~None
- Category: Coding Practices [2]
- CWE subcategory: CWE-1041 [1]

Description

If there are multiple sidechains **freezeTokenForOtherChainAsset** may not be able to specify which chain of **NFT** assets is being lent.

```
    function freezeTokenForOtherChainAsset(
        address game,
        address holder,
        uint internalId,
        uint amount,
        bool useCredit
    ) public whenNotPaused {
        require(isMainChain, "only freeze on main chain");
        bool canBorrow = ValidateLogic.checkFreezeForOtherChainPara(game, internalId,
            freezeMap);
        require(canBorrow, "can't borrow asset now");
```

Listing 3.4: KyokoCCAL.sol

Kyoko-CCAL Reply:

We have marked chainid in the front-end display, which is ignored here.

3.4 Risks of whenNotPaused

- ID: MSVE-004
- Severity: Low
- Likelihood: N/A
- Impact: N/A
- Target: `~None`
- Category: Coding Practices [2]
- CWE subcategory: CWE-1041 [1]

Description

The **repayAsset** and **syncInterestAfterRepayViaBot** methods have **whenNotPaused**.
It may cause the borrower to pay high interest or even overdue because the project side temporarily suspends the function.

```
    function repayAsset(address game, address holder, uint internalId) public
        whenNotPaused {

        ValidateLogic.checkRepayAssetPara(game, _msgSender(), internalId, nftMap);

        DepositTool storage asset = nftMap[internalId];

        for (uint idx; idx < asset.toolIds.length; idx++) {
            IERC721Upgradeable(game).safeTransferFrom(_msgSender(), address(this), asset
                .toolIds[idx]);
        }
```

Listing 3.5: KyokoCCAL.sol

```
    function syncInterestAfterRepayViaBot(
        address game,
        address holder,
        uint internalId,
        uint interest
    ) public whenNotPaused onlyBot {
        pendingWithdrawInterest[holder].push(
            InterestInfo({
                internalId: internalId,
                amount: interest,
                game: game
            })
        );
```

Listing 3.6: KyokoCCAL.sol

Kyoko-CCAL Reply:

repayAsset and syncInterestAfterRepayViaBot methods have been removed whenNotPaused

Status:

Fixed.

4 Conclusion

In this audit, we have analyzed the Kyoko-CCAL 's contracts design and implementation.
During this audit, we did not find any exploitable security issues. And at the same time, we also paid attention to some minor issues when building the code and communicated with Kyoko-CCAL. They are very familiar with the code business logic and business scenarios, which brought great convenience to our audit during the communication process. After we found some problems, they ignored two of them and gave a reply, and we responded to the Agreed. Ignoring states does not imply additional security risks. After audit, the code level and economic model without security issues.

As a final precaution, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.2 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [5, 6, 7, 8, 9].
- Result: Not found
- Severity: Critical

5.1.3 Reentrancy

- Description: Reentrancy [10] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.4 Parameters Injection

- Description: Injected parameters can be malicious and introduce sever vulnerability if no input sanitization is used.
- Result: Not found

- Severity: Critical

5.1.5 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.6 DoS

- Description: Whether the contract is vulnerable to DoS attack.
- Result: Not found
- Severity: Medium

5.1.7 Random Number Security

- Description: Correctly generated random number is the foundation of the smart contracts.
- Result: Not found
- Severity: Critical

5.1.8 Delegate Call Security

- Description: delegate_call should be used to limit the proxy address.
- Result: Not found
- Severity: Critical

5.1.9 Timestamp Dependency Attack

- Description: Incorrect timestamp can cause rewards lost.
- Result: Not found
- Severity: Medium

5.1.10 False Top-up

- Description: False Top-up can be caused by a delay transaction and results in financial lost .
- Result: Not found
- Severity: Medium

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[3] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[4] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[5] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[6] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[7] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[8] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[9] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[10] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.