SECURITY AUDIT REPORT

for

KYOKO-G2G

Prepared By: yeye

January 20, 2022

Document Properties

Client	Kyoko-G2G
Title	Security Audit Report
Target	Ethereum Smart Contract
Version	1.1
Author	yeye
Auditors	Darker, Haoqi,Ye ye
Reviewed by	Haoqi
Approved by	yeye
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	January 20, 2022	yeye	Perview Release
final	January 27, 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-G2G

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-G2G

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

Kyoko.finance-G2G is mainly a lending agreement for DAO and ordinary users, ordinary users and DAO can deposit stable coins to get interest, DAO can make unsecured loans (or rent game assets) with the credit line.

Users can deposit money (stable coins such as **USDT**) into **LendingPool** to get interest, and the deposit will mint the corresponding amount of **kToken**; designated DAO (containing credit line, credit line is set by multi-signature wallet review and approval) can borrow **USDT** (a stable coins) or game assets through **LendingPool**, the interest of both is mainly through The liquidity of the funds in the **LendingPool** pool is controlled.

The interest rate on the lending side uses a compound interest scheme and the interest rate on the deposit side uses a linear scheme. The interest rate on the credit side is divided into two gradients, **slope1** and **slope2**, and the interest rate is calculated using slope1 for funds within the optimal capital utilization rate, and the interest rate is calculated using the compound operation of **slope1** and **slope2** for funds above the optimal utilization rate.

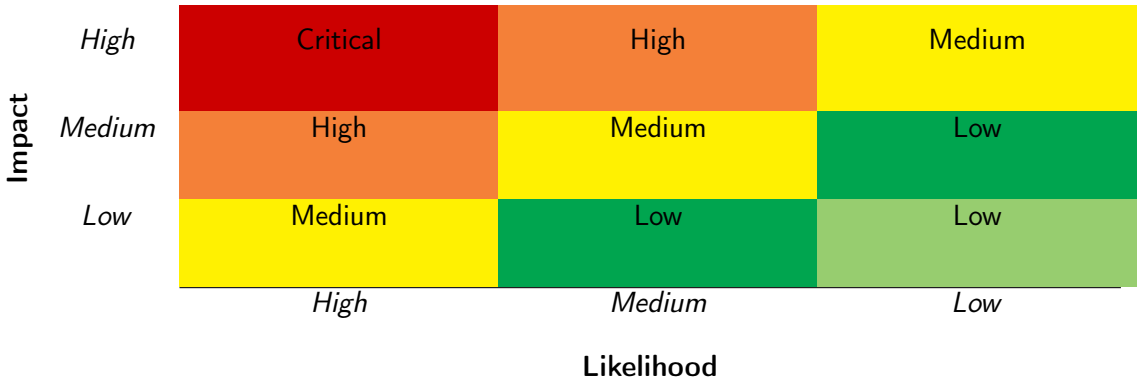
There are multiple assets in **LendingPool** corresponding to Reserve, and only in the case of generating loans, the user's deposit will have interest, while the higher the utilization rate of funds, the higher the interest rate of loans, and the higher the interest rate of corresponding deposits. deposits and loans in **LendingPool** are normalized in the system, uniformly normalized to the T0 moment. Each user's deposit, withdraw, borrow and repay operations will update the Reserve information and the deposit and loan interest rates.

The basic information of Ethereum Smart Contract is as follows:

Table 1.1: Basic Information of Ethereum Smart Contract

Item	Description
Issuer	Kyoko-G2G
Website	https://www.kyoko.finance/
Type	Ethereum Smart Contract
Platform	solidity
Audit Method	Whitebox
Latest Audit Report	January 20, 2022

Table 1.2: Vulnerability Severity Classification



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.

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.

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

- 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-G2G 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	0	
Informational	4	
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	Info	Decimals over 18 are not supported	Coding Practices	Informed
MSVE-002	Info	AmountSent should be !=type(uint256).max	Coding Practices	Fixed
MSVE-003	Info	Timelock is recommended to be added	Coding Practices	Informed
MSVE-004	Info	Contracts is recommended for investment and voting	Coding Practices	Informed

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 Decimals over 18 are not supported

- ID: MSVE-001
- Severity: info
- Likelihood: N/A
- Impact: N/A
- Target: `vars.tokenUnit`
- Category: Coding Practices [2]
- CWE subcategory: CWE-1041 [1]

Description

Tokens with decimals over 18 are not supported, so be careful when adding new tokens.
In addition there are basically no stable coin decimals over 18.
Just a reminder that,the issue can somehow be ignored.

It calculates the debit amount uniformly converted to 1e18, if the decimals are greater than 18 after conversion is 0.

```
76    {
77          CalculateUserAccountDataVars memory vars;
78          for (vars.i = 0; vars.i < reservesCount; vars.i++) {

80              vars.currentReserveAddress = reserves[vars.i];
81              DataTypes.ReserveData storage currentReserve = reservesData[vars.
                    currentReserveAddress];

83              vars.decimals = currentReserve.getDecimal();
84              uint256 decimals_ = 1 ether;
85              vars.tokenUnit = uint256(decimals_).div(10**vars.decimals);
```

Listing 3.1: GenericLogic.sol

3.2 AmountSent should be !=type(uint256).max

- ID: MSVE-002
- Severity: info
- Likelihood: N/A
- Impact: N/A
- Target: `amountSent`
- Category: Coding Practices [2]
- CWE subcategory: CWE-1041 [1]

Description

AAVE contract version is 0.6.12 written as:amountSent ! = uint256(-1).
After **Kyoko-G2G** changed it, they upgraded the version number to 0.8.7, which does not support the syntax uint256(-1).

So uint256(-1) should be type(uint256).max, not min.
Just a reminder that,the issue can somehow be ignored.

```
97      function validateRepay(
98          DataTypes.ReserveData storage reserve,
99          uint256 amountSent,
100         address onBehalfOf,
101         uint256 variableDebt
102     ) external view {
103         bool isActive = reserve.getActive();

105         require(isActive, "VL_NO_ACTIVE_RESERVE");

107         require(amountSent > 0, "VL_INVALID_AMOUNT");

109         require(variableDebt > 0, "VL_NO_DEBT_OF_SELECTED_TYPE");

111         require(
112             amountSent != type(uint256).min  msg.sender == onBehalfOf,
113             "VL_NO_EXPLICIT_AMOUNT_TO_REPAY_ON_BEHALF"
114         );
115     }
116 }
```

Listing 3.2: ValidationLogic.sol

3.3 Timelock 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

After our analysis of the contract, there are potential security risks in source of multi-signature accounts and whitelist accounts management, in order to prevent the risk of collusion, so we recommend adding a layer of time lock in addition to multi-signature, and set up event monitoring to further enhance the project security.

3.4 Contracts is recommended for investment and voting

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

Description

Investment Strategies and Governance also suggest using contracts for investing and voting as well, as a way to increase trust in the agreement.
Because the business logic is such that users cannot borrow, they can only deposit and withdraw; the code for deposits and withdrawals is basically the same as the **AAVE** project.
In addition, because the borrowing was changed to only **whitelist** addresses can borrow, so the security of funds depends mainly on how the **whitelist** addresses operate, and if the funds are borrowed and then not returned, it is the same as a loss of funds.
So the last is to suggest that this **whitelist** address is best an investment strategy contract, as is the same practice with **DeFi** vault.

4 Conclusion

In this audit, we have analyzed the Kyoko-G2G 's contracts design and implementation. The project mainly refers to **AAVE V2** for development, the developer is to understand the AAVE source code in-depth modification, the main changes are: DAO can be unsecured credit model, and remove the **stableRate**, **flashlon** and other functions, simplify a lot of code. 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.