SMART CONTRACT AUDIT REPORT

for

Shinkai Protocol

Prepared By: Xiaomi Huang

PeckShield

January 2, 2023

Document Properties

Client	Shinkai
Title	Smart Contract Audit Report
Target	Shinkai
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 2, 2023	Xuxian Jiang	Final Release
1.0-rc1	December 5, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Shinkai protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the audited protocol can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Shinkai

Shinkai allows user to register an identity (e.g. nico.shinkai) by staking certain amount of SHIN tokens. The required stake amount is computed proportionally inverse to the length of identity name. A user may stake more than what is needed to buy that specific identity, and should be able to partially withdraw that extra difference without losing the identity. The identity registration requires submitting an address that will be the owner of the identity. If the user chooses to unstake SHIN tokens, it may simply lose the ownership of the identity. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Shinkai

Item	Description
Name	Shinkai
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 2, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/dcSpark/shinkai-contracts.git (fa9517f)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/dcSpark/shinkai-contracts.git (01c1f58)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items 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 testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC 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) [7], 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.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, 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 Logics	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 `Shinkai` 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	1	■
Low	2	■■
Informational	0	
Total	3	

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 that 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 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Shinkai Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved claimIdentityBatched() Logic in ShinkaiRegistry	Business Logic	Resolved
PVE-002	Low	Revisited _setRecord() Logic in ShinkaiRegistry	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Improved claimIdentityBatched() Logic in ShinkaiRegistry

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ShinkaiRegistry`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

The `Shinkai` protocol allows for identity registration by staking required `SHIN` tokens. While reviewing the batch logic to claim multiple identities, we notice the implementation can be improved by validating no duplicate in the given identities.

To elaborate, we show below the code snippet of the `claimIdentityBatched()` routine, which is used to claim multiple identities altogether. Each claimed identity will be validated with a helper `_validateClaim()`, which will ensure the claimed identity does not have an owner yet. Since the routine handles a batch of identities, it is possible a new claimed identity may not have an owner yet, but is also claimed in earlier batched processing. In other words, the routine may be improved to ensure no duplicate will be allowed in the user input.

```
138     function claimIdentityBatched(
139         string[] calldata names,
140         uint256[] calldata namespaces,
141         uint256[] calldata stakeAmounts,
142         address[] calldata owners
143     ) public {
144         if (names.length != namespaces.length  names.length != stakeAmounts.length
                names.length != owners.length) {
145             revert InputArityMismatch();
146         }
147         uint256 startTokenId = shinkaiNft.nextTokenId();
148         address prevOwner = owners[0];
149         uint256 nftTokensToMint = 0;
```

```
150         uint256 totalPayment;
151         _updateRewardsState();
152         for (uint256 i = 0; i < names.length; i++) {
153             if (owners[i] != prevOwner) {
154                 shinkaiNft.mint(prevOwner, nftTokensToMint);
155                 nftTokensToMint = 1;
156                 prevOwner = owners[i];
157             } else {
158                 nftTokensToMint++;
159             }
160             if (i == names.length - 1) {
161                 shinkaiNft.mint(prevOwner, nftTokensToMint);
162             }
163
164             (string memory identity, uint256 payment) = _validateClaim(names[i],
                    namespaces[i], stakeAmounts[i]);
165             totalPayment += payment;
166             identityRecords[identity].stakedTokens = payment;
167             emit StakeUpdate(identity, payment);
168             _setBoundNft(identity, startTokenId + i);
169             uint256 rewardsStateIndex = uint256(rewardsState.index);
170             identityStakingIndex[identity] = rewardsStateIndex;
171             identityDelegationIndex[identity] = rewardsStateIndex;
172         }
173         shinToken.transferFrom(msg.sender, address(this), totalPayment);
174     }
```

Listing 3.1: ShinkaiRegistry :: claimIdentityBatched()

In addition, the above routine may be improved by relocating the following statement `uint256 rewardsStateIndex = uint256(rewardsState.index);` (line 169) outside for-loop to avoid repeated storage reads.

Recommendation Revise the above routine to avoid duplicate identity claims.

Status This issue has been fixed in the following PR: 6.

3.2 Revisited _setRecord() Logic in ShinkaiRegistry

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ShinkaiRegistry`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The ShinkaiRegistry contract has a helper routine to allow for the owner to set various records, including keys and routing addresses. Our analysis on the routine shows the current implementation can be improved.

To elaborate, we show below the implementation of the related _setRecord routine. We notice the use of emptyStringHash to reset current identity keys. However, the emptyStringHash is computed as emptyStringHash = keccak256(bytes(params.encryptionKey)) (line 643), which does not make use of an empty string for the hash computation.

```
642     function _setRecord(string memory identity, SetRecordParams calldata params)
            internal {
643         bytes32 emptyStringHash = keccak256(bytes(params.encryptionKey));
644         if (
645             keccak256(bytes(params.encryptionKey)) == emptyStringHash
646                 && keccak256(bytes(params.signatureKey)) == emptyStringHash
647         ) {
648             _unsetKeys(identity);
649         } else {
650             _setKeys(identity, params.encryptionKey, params.signatureKey);
651         }
652         if (params.addressOrProxyNodes.length == 0) {
653             _unsetaddressOrProxyNodes(identity);
654         } else {
655             if (params.routing) {
656                 _setProxyNodes(identity, params.addressOrProxyNodes);
657             } else {
658                 _setNodeAddress(identity, params.addressOrProxyNodes[0]);
659             }
660         }
661     }
```

Listing 3.2: ShinkaiRegistry::_setRecord()

Recommendation Revise the above routine to make use of the intended empty string hash calculation.

Status This issue has been fixed in the following PR: 7.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: ShinkaiRegistry
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the Shinkai protocol, there is a privileged account (owner). This account plays critical roles in governing and regulating the protocol-wide operations (e.g., configure protocol parameters and upgrade protocol implementations). Our analysis shows that the privileged account needs to be scrutinized. In the following, we use the ShinkaiRegistry contract as an example and show the representative functions potentially affected by the privileged account.

```
580     function setBaseRewardsRate(uint256 rate) public onlyOwner {
581         if (rate > baseRewardsRateMaxMantissa) {
582             revert InvalidBaseRewardsRate(rate);
583         }
584         _updateRewardsState();
585         baseRewardsRate = rate;
586         emit BaseRewardsRateUpdate(rate);
587     }

589     function _authorizeUpgrade(address) internal override onlyOwner {}
```

Listing 3.3: Example Privileged Operations in ShinkaiRegistry

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the administrative privileges to the intended DAO-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team confirmed that all the privileged accounts will be multi-sig wallets.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Shinkai protocol, which allows user to register an identity (e.g. nico.shinkai) by staking certain amount of SHIN tokens. The required stake amount is computed proportionally inverse to the length of identity name. A user may stake more than what is needed to buy that specific identity, and should be able to partially withdraw that extra difference without losing the identity. The identity registration requires submitting an address that will be the owner of the identity. If the user chooses to unstake SHIN tokens, it may simply lose the ownership of the identity. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.