SMART CONTRACT AUDIT REPORT

for

RockX Liquid Staking (IoTeX)

Prepared By: Xiaomi Huang

PeckShield

September 1, 2023

Document Properties

Client	RockX
Title	Smart Contract Audit Report
Target	RockX Liquid Staking (IoTeX)
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	September 1, 2023	Xuxian Jiang	Final Release
1.0-rc	August 31, 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 source code of the `Liquid Staking (IoTeX)` support in `RockX`, 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 given version of smart contracts 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 RockX

`RockX` is a blockchain fintech company that helps users embrace `Web 3.0` effortlessly through the development of innovative products and infrastructure. It also strives to enable institutions and disruptors in the financial and Internet sectors to gain seamless access to blockchain data, crypto yield products and best-in-class key management solutions in a sustainable way. The audited `Liquid Staking (IoTeX)` component allows users to stake `IOTX` and vote for `delegates` to support and expand the network. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The RockX Liquid Staking (IoTeX)

Item	Description
Name	RockX
Website	https://www.rockx.com/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 1, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers all files under the `contracts/` directory.

- https://github.com/RockX-SG/uniiotx.git (1d161a9)

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

- https://github.com/RockX-SG/uniiotx.git (6cbef40)

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 implementation of the `Liquid Staking (IoTeX)` component in `RockX`. 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 logics, 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. 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 RockX Liquid Staking (IoTeX) Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence of Checks-Effects-Interactions	Coding Practices	Resolved
PVE-002	Low	Possible Costly uniIOTX From Improper Staking Initialization	Time and State	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IOTXClear`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [11] exploit, and the `Uniswap/Lendf.Me` hack [10].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `IOTXClear` as an example, the `claimRewards()` function (see the code snippet below) is provided to externally call a token contract to claim available rewards. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 444) start before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
436     function claimRewards(uint amount, address recipient) external nonReentrant
            whenNotPaused {
437         // Update reward
438         _updateUserReward(msg.sender);
```

```

        // Check reward
        UserInfo storage info = userInfos[msg.sender];
        require(info.reward >= amount, "USR005"); // Insufficient accounted reward

        // Transfer reward
        payable(recipient).sendValue(amount);
        info.reward -= amount;
        accountedBalance -= amount;

        emit RewardClaimed(msg.sender, recipient, amount);
    }
```

Listing 3.1: `IOTXClear::claimRewards()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`. Also, the current functions have the associated `nonReentrant` modifier, which eliminate the issue. However, we still feel the need of following the best practice of `checks-effects-interactions`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle to block possible `re-entrancy`.

Status The issue has been addressed by the following commit: `12fdbfe`.

3.2 Possible Costly uniIOTX From Improper Staking Initialization

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IOTXStaking`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

The `RockX Liquid Staking` component allows users to stake `IOTX` and vote for `Delegates` to support and expand the network. The deposit of supported assets will get in return the share to represent the pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `_mint()` routine, which is used for participating users to deposit the supported assets and get respective pool shares in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function _mint() internal returns (uint minted) {
        accountedBalance += msg.value;

        uint toMint = _convertIotxToUniIOTX(msg.value);
        IUniIOTX(uniIOTX).mint(msg.sender, toMint);
        minted = toMint;

        totalPending += msg.value;

        emit Minted(msg.sender, minted);
    }
```

Listing 3.2: `IOTXStaking::_mint()`

```
765     function _convertIotxToUniIOTX(uint amountIOTX) internal view returns (uint
            uniIOTXAmount) {
766         uint totalSupply = IUniIOTX(uniIOTX).totalSupply();
767         uint currentReserveAmt = _currentReserve();
768         uniIOTXAmount = DEFAULT_EXCHANGE_RATIO * amountIOTX;

770         if (currentReserveAmt > 0) { // avert division overflow
771             uniIOTXAmount = totalSupply * amountIOTX / currentReserveAmt;
772         }
773     }
```

Listing 3.3: `IOTXStaking::_convertIotxToUniIOTX()`

Specifically, when the pool is being initialized (line 768), the share value directly takes the value of `amountIOTX` (line 768), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `uniIOTXAmount = DEFAULT_EXCHANGE_RATIO * amountIOTX = 1 WEI`. With that, the actor can further deposit a huge amount of the underlying assets (right before the `_syncReward()` call) with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current deposit logic to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been addressed by the following commit: `5b5fa45`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In `RockX Liquid Staking` support, there is a privileged administrative account (the account with the `DEFAULT_ADMIN_ROLE` role). The administrative account plays a critical role in governing and regulating the staking-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `IOTXStaking` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

Specifically, the privileged functions in `IOTXStaking` allow for the `DEFAULT_ADMIN_ROLE` role to assign the `ROLE_ORACLE` role to configure `globalDelegate`, update `tokenID` delegates, initiate stake operations etc.

```
    function setGlobalDelegate(address delegate) external whenNotPaused onlyRole(
        ROLE_ORACLE) {
        globalDelegate = delegate;

        emit GlobalDelegateSet(delegate);
    }

    /**
     * @dev This function updates the delegates of token IDs.
     */
    function updateDelegates(uint[] calldata tokenIds, address delegate) external
        whenNotPaused onlyRole(ROLE_ORACLE) {
        ISystemStaking(systemStaking).changeDelegates(tokenIds, delegate);

        emit DelegatesUpdated(tokenIds, delegate);
    }

    /**
     * @dev This function stakes any pending IOTXs and merges staked buckets when
         conditions are fulfilled.
     */
    function stake() external whenNotPaused onlyRole(ROLE_ORACLE) {
```

```
        _stakeAtTopLevel();
        _stakeAndMergeAtSubLevel();
    }
```

Listing 3.4: Example Privileged Operations in `IOTXStaking`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirms that all the privileged roles will be transferred to `Gnosis multi-sig`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Liquid Staking (IoTeX)` component in `RockX`, which allows users to stake `IOTX` and vote for `delegates` to support and expand the network. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.