SMART CONTRACT AUDIT REPORT

for

Rentable Protocol

Prepared By: Xiaomi Huang

PeckShield

June 2 2022

Document Properties

Client	Rentable
Title	Smart Contract Audit Report
Target	Rentable
Version	1.1
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	June 2, 2022	Xuxian Jiang	Post-Final Revision #1
1.0	April 23, 2022	Xuxian Jiang	Final Release
1.0-rc1	April 22, 2022	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 `Rentable` 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 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 Rentable

`Rentable` is an `NFT` renting protocol that allows the holders of `NFT` collection (e.g., `Decentraland LAND`) to deposit into the protocol and get an `ORentable` token to represent the ownership. Each `NFT` collection also has a respective `WRentable` with the same token id, which is minted when rental starts and burnt on expiry. The renter will pay the required rent fee to the respective rentee. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Rentable` Protocol

Item	Description
Issuer	Rentable
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 2 2022

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

- https://github.com/rentable-world/rentable-protocol-private.git (9c79833)

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

- https://github.com/rentable-world/rentable-protocol-private.git (95d6ca3)

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

Likelihood

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.

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

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

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 design and implementation of the `Rentable` protocol. 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	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 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, 2 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Rentable Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improper Operator Handover Upon WToken Transfers of Expired Rents	Business Logic	Resolved
PVE-002	Low	Improved Validation on Protocol Fee Changes	Coding Practices	Resolved
PVE-003	Medium	Trust on Admin Keys	Security Features	Confirmed
PVE-004	Informational	Meaningful Events For Important States Change	Coding Practices	Resolved

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 Improper Operator Handover Upon WToken Transfers of Expired Rents

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Rentable protocol manages the deposited NFT collections with the associated ORentable and WRentable to keep track of the current rentee as well as the potential renter. The transfers of the associated ORentable and WRentable will require the necessary updates due to the rentee/renter changes. While examining the logic behind the WRentable transfer, we notice an issue that may unnecessarily grant extra roles to the recipient.

To elaborate, we show below the afterWTokenTransfer() routine, which is invoked whenever the WRentable token is transferred. This routine has a rather straightforward logic in examining whether the rent is expired. If expired, the underlying token's update role is properly transferred back to the current ORentable owner. However, the current implementation has the logic in invoking the delegatecall of postWTokenTransfer(), which grants the WRentable recipient with the underlying token's update role – even if the WRentable is now expired!

```
    function afterWTokenTransfer(
        address tokenAddress,
        address from,
        address to,
        uint256 tokenId
    ) external override whenNotPaused onlyWToken(tokenAddress) {
        _expireRental(address(0), tokenAddress, tokenId, true);

        address lib = _libraries[tokenAddress];
```

```
        if (lib != address(0)) {
            // slither-disable-next-line unused-return
            lib.functionDelegateCall(
                abi.encodeWithSelector(
                    ICollectionLibrary.postWTokenTransfer.selector,
                    tokenAddress,
                    tokenId,
                    from,
                    to
                ),
                ""
            );
        }
    }
```

Listing 3.1: `Rentable::afterWTokenTransfer()`

To correct, there is a need to check the current `rent` status. If the `rent` becomes expired, there is a need to inform the subsequent `postWTokenTransfer()` call so that it should avoid granting the update role to the recipient.

Recommendation Revise the transfer logic of `WRentable` to properly manage the update role of the underlying `NFT` collections.

Status The issue has been fixed by this commit: `4613d41`.

3.2 Improved Validation on Protocol Fee Changes

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Rentable`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Rentable` protocol is no exception. Specifically, if we examine the `Rentable` contract, it has defined a number of protocol-wide risk parameters, such as `_fee` and `_paymentTokenAllowlist`. In the following, we show the corresponding routines that allow for their changes.

```
    function setFee(uint16 newFee) external onlyGovernance {
        uint16 previousFee = _fee;

        _fee = newFee;

```

```
        emit FeeChanged(previousFee, newFee);
    }

    /// @dev Set fee collector address
    /// @param newFeeCollector fee collector address
    function setFeeCollector(address payable newFeeCollector)
        external
        onlyGovernance
    {
        require(newFeeCollector != address(0), "FeeCollector cannot be null");

        address previousFeeCollector = _feeCollector;

        _feeCollector = newFeeCollector;

        emit FeeCollectorChanged(previousFeeCollector, newFeeCollector);
    }

    /// @dev Enable payment token (ERC20)
    /// @param paymentToken payment token address
    function enablePaymentToken(address paymentToken) external onlyGovernance {
        uint8 previousStatus = _paymentTokenAllowlist[paymentToken];

        _paymentTokenAllowlist[paymentToken] = ERC20_TOKEN;

        emit PaymentTokenAllowListChanged(
            paymentToken,
            previousStatus,
            ERC20_TOKEN
        );
    }
```

Listing 3.2: Rentable::setFee() and Rentable::setFeeCollector()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `_fee` may charge unreasonably high fee in the `rent` payment, hence incurring cost to borrowers or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been fixed by this commit: `4613d41`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Rentable`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Rentable` protocol, there is a special administrative account, i.e., `governance`. This `governance` account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings, pause/unpause the protocol, as well as update the proxy implementation). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setLibrary(address tokenAddress, address libraryAddress)
        external
        onlyGovernance
    {
        address previousValue = _libraries[tokenAddress];

        _libraries[tokenAddress] = libraryAddress;

        emit LibraryChanged(tokenAddress, previousValue, libraryAddress);
    }

    /// @dev Associate the otoken to the specific wrapped token
    /// @param tokenAddress wrapped token address
    /// @param oRentable otoken address
    function setORentable(address tokenAddress, address oRentable)
        external
        onlyGovernance
    {
        address previousValue = _orentables[tokenAddress];

        _orentables[tokenAddress] = oRentable;

        emit ORentableChanged(tokenAddress, previousValue, oRentable);
    }

    /// @dev Associate the otoken to the specific wrapped token
    /// @param tokenAddress wrapped token address
    /// @param wRentable otoken address
    function setWRentable(address tokenAddress, address wRentable)
        external
        onlyGovernance
    {
```

```
        address previousValue = _wrentables[tokenAddress];

        _wrentables[tokenAddress] = wRentable;

        emit WRentableChanged(tokenAddress, previousValue, wRentable);
    }
```

Listing 3.3: Example Privileged Operations in `Rentable`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these 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 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 The issue has been confirmed by the team. The team clarifies that the `governance` account will be managed by a multi-sig account.

3.4 Generation of Meaningful Events For Important State Changes

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `WalletFactory`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `WalletFactory` contract as an example. This contract has a public function that is used to update the `beacon`. While examining the event that reflect the `beacon` change, we notice there is a lack of emitting the respective event (line 39).

```
    /// @dev Set beacon for new wallets
    /// @param beacon beacon address
    function _setBeacon(address beacon) internal {
        // it's ok to se to 0x0, disabling factory
        // slither-disable-next-line missing-zero-check
        _beacon = beacon;
    }

    /* ---------- Public ---------- */

    /// @notice Set beacon for new wallets
    /// @param beacon beacon address
    function setBeacon(address beacon) external onlyOwner {
        _setBeacon(beacon);
    }
```

Listing 3.4: `WalletFactory::setBeacon()`

Recommendation Properly emit the respective event `WalletBeaconChanged` when a new `beacon` becomes effective.

Status This issue has been fixed in the following commit: `95d6ca3`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Rentable` protocol, which is an `NFT` renting protocol that allows the holders of `NFT` collection (e.g., `Decentraland LAND`) to deposit and get an `ORentable` token to represent the ownership. Each `NFT` collection also has a respective `WRentable` with the same token id, which is minted when rental starts and burnt on expiry. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.