SMART CONTRACT AUDIT REPORT

for

Chee Finance Lend

Prepared By: Xiaomi Huang

PeckShield

May 26, 2022

Document Properties

Client	Chee Finance
Title	Smart Contract Audit Report
Target	Chee Finance Lend
Version	1.0-rc
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	May 26, 2022	Xiaotao Wu	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 `Chee Finance Lend` feature, 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 Chee Finance Lend

The `Chee Finance Lend` feature is designed for users to deposit `veNFTs`, and mint `cheeTokens` accordingly. The amount of `cheeTokens` minted will be determined by the locked amount of tokens in `veNFT`. When paying back, the borrower will need to pay back the same amount of `cheeTokens` and the client's tokens as interest. There is also a liquidation function that can be invoked when the borrow position is expired (and after grace period) and the position can be liquidated. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Chee Finance Lend

Item	Description
Issuer	Chee Finance
Website	https://www.chee.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 26, 2022

In the following, we show the MD5 hash value of the related compressed file with the contract for audit:

- MD5 (Lend.zip) = 732b8127ae94507185e2d9793c33820b

And this is the MD5 checksum value of the compressed file after fixes for the main issues found in the audit have been checked in:

- MD5 (Lend_fix.zip) = 9c445f354c9c4105e1f42df5c06daa36

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 `Chee Finance Lend` feature 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 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	2	■■
Low	0	
Undetermined	1	■
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 2 medium-severity vulnerabilities and 1 undetermined issue.

Table 2.1: Key Chee Finance Lend Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Interest Calculation Logic in Lend::repay()	Business Logic	Fixed
PVE-002	Undetermined	Potential Reentrancy Risks In Lend::repay()	Time and State	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Incorrect Interest Calculation Logic in Lend::repay()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Vaults`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Lend` contract of `Chee Finance` provides an external `repay()` function for users to pay back the borrowed `cheeTokens` and get back their deposited `veNFTs`. Our analysis with this routine shows the current calculation logic for the accumulated interest is not correct.

To elaborate, we show below its code snippet. Specifically, the state variable `userInfo[_veNftAddress][_nftId].lockPercentage` is used for calculating the `interest` (lines 186-187). However, this variable is defined but never assigned, thus the calculated result for the `interest` will always equal to 0.

```
175     function repay(
176         address _veNftAddress,
177         uint256 _cheeAmount,
178         uint256 _nftId
179     ) public {
180         require(_veNftAddress != address(0) && _cheeAmount > 0 && _nftId > 0, "Incorrect
             Input");
181         require(IERC721(_veNftAddress).ownerOf(_nftId) == address(this), "Nft Id not
            detected in contract");
182         require(userInfo[_veNftAddress][_nftId].lender == msg.sender, "You have not
            lender of this nft");
183         uint256 lockPeriod = userInfo[_veNftAddress][_nftId].lockPeriod.add(
            maxLendingPeriod).add(graceTime);
184         require(lockPeriod >= block.number, "Cannot repay Now");
185         require(userInfo[_veNftAddress][_nftId].amount == _cheeAmount, "Enter Correct
            Amount");
186         value = (IVeNft(_veNftAddress).getAmount(_nftId).mul(userInfo[_veNftAddress][
            _nftId].lockPercentage)).div(100);
```

```
        uint256 interest = (value.mul(block.number.sub(userInfo[_veNftAddress][_nftId].lockPeriod)).mul(interestRatePerBlock)).div(100);
        IChee(supportedAddress1[_veNftAddress]).burnFrom(msg.sender, _cheeAmount);
        IToken(supportedAddress2[_veNftAddress]).transferFrom(msg.sender, depositAddress, interest);
        IVeNft(_veNftAddress).safeTransferFrom(address(this), msg.sender, _nftId);
    }
```

Listing 3.1: `Lend::repay()`

Recommendation Correctly calculate the accumulated `interest`. An example revision is shown below:

```
    function repay(
        address _veNftAddress,
        uint256 _cheeAmount,
        uint256 _nftId
    ) public {
        require(_veNftAddress != address(0) && _cheeAmount > 0 && _nftId > 0, "Incorrect Input");
        require(IERC721(_veNftAddress).ownerOf(_nftId) == address(this), "Nft Id not detected in contract");
        require(userInfo[_veNftAddress][_nftId].lender == msg.sender, "You have not lender of this nft");
        uint256 lockPeriod = userInfo[_veNftAddress][_nftId].lockPeriod.add(maxLendingPeriod).add(graceTime);
        require(lockPeriod >= block.number, "Cannot repay Now");
        require(userInfo[_veNftAddress][_nftId].amount == _cheeAmount, "Enter Correct Amount");
        uint256 interest = (_cheeAmount.mul(block.number.sub(userInfo[_veNftAddress][_nftId].lockPeriod)).mul(interestRatePerBlock)).div(100);
        IChee(supportedAddress1[_veNftAddress]).burnFrom(msg.sender, _cheeAmount);
        IToken(supportedAddress2[_veNftAddress]).transferFrom(msg.sender, depositAddress, interest);
        IVeNft(_veNftAddress).safeTransferFrom(address(this), msg.sender, _nftId);
    }
```

Listing 3.2: `Lend::repay()`

Status This issue has been fixed.

3.2 Potential Reentrancy Risks In Lend::repay()

- ID: PVE-002
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Lend`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [2]

Description

As mentioned in Section 3.1, the `repay()` function in the `Lend` contract is provided for users to pay back the borrowed `cheeTokens` and get back their deposited `veNFTs`. While reviewing the current `Lend` contract, we notice there is a potential reentrancy risk in current implementation.

To elaborate, we show below the code snippet of the `repay()` routine in `Lend`. The execution logic is rather straightforward: after the borrower pays back the `cheeTokens` and the client's tokens as interest, it transfers the `veNFT` from the `Lend` contract to the borrower. However, our analysis shows that the current implementation of `claim()` can be improved for re-entrancy prevention.

Specifically, when the `safeTransferFrom()` action occurs, the `onERC721Received()` function will be called on the recipient contract. Consequently, any `safeTransferFrom()` of ERC721-based tokens might introduce the chance for reentrancy to execute for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in `IVeNft(_veNftAddress).safeTransferFrom()` (line 190). So far, we also do not know how an attacker can exploit this issue to earn profit. After internal discussion, we consider it is necessary to bring this issue up to the team. Though the implementation of the `repay()` function is well designed, we may intend to use the `ReentrancyGuard::nonReentrant` modifier to protect the `repay()` function at the whole protocol level.

```
175     function repay(
176         address _veNftAddress,
177         uint256 _cheeAmount,
178         uint256 _nftId
179     ) public {
180         require(_veNftAddress != address(0) && _cheeAmount > 0 && _nftId > 0, "Incorrect
             Input");
181         require(IERC721(_veNftAddress).ownerOf(_nftId) == address(this), "Nft Id not
            detected in contract");
182         require(userInfo[_veNftAddress][_nftId].lender == msg.sender, "You have not
            lender of this nft");
183         uint256 lockPeriod = userInfo[_veNftAddress][_nftId].lockPeriod.add(
            maxLendingPeriod).add(graceTime);
184         require(lockPeriod >= block.number, "Cannot repay Now");
185         require(userInfo[_veNftAddress][_nftId].amount == _cheeAmount, "Enter Correct
            Amount");
186         value = (IVeNft(_veNftAddress).getAmount(_nftId).mul(userInfo[_veNftAddress][
            _nftId].lockPercentage)).div(100);
```

```
        uint256 interest = (value.mul(block.number.sub(userInfo[_veNftAddress][_nftId].
            lockPeriod)).mul(interestRatePerBlock)).div(100);
        IChee(supportedAddress1[_veNftAddress]).burnFrom(msg.sender, _cheeAmount);
        IToken(supportedAddress2[_veNftAddress]).transferFrom(msg.sender, depositAddress
            , interest);
        IVeNft(_veNftAddress).safeTransferFrom(address(this), msg.sender, _nftId);
    }
```

Listing 3.3: `Lend::repay()`

Note a similar issue also exists in the `liquidate()` routine of the same contract.

Recommendation Apply the non-reentrancy protection in the above-mentioned routine.

Status This issue has been fixed.

3.3 Trust Issue of Admin Keys

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

Description

In the `Chee Finance Lend` feature, there is a privileged account, i.e., `owner`. The `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., configure key parameters, execute privileged operations, etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the `owner` account.

```
    function changeAdmin(address _newAdmin) public onlyAdmin {
        require(_newAdmin != address(0), "Zero Address Detected");
        admin = _newAdmin;
      }

      function whitelist(address _user) public onlyAdmin {
        require(_user != address(0), "Zero Address Detected");
        permissioned[_user] = true;
      }

      function setInterestRatePerBlock(uint256 _interestRate) public onlyAdmin {
        require(_interestRate > 0, "Value cannot be less then 1");
        interestRatePerBlock = _interestRate;
      }

      function setFeeRate(uint256 _feeRate) public onlyAdmin {
```

```
        require(_feeRate > 0, "Value cannot be less then 1");
        feeRate = _feeRate;
    }

    function setCollateralPercentageThreshold(uint256 _maximumLockPercentage) public onlyAdmin {
        require(_maximumLockPercentage > 0, "Value cannot be less then 1");
        lockPercentageThreshold = _maximumLockPercentage;
    }

    function setGraceTime(uint256 _timeInBlocks) public onlyAdmin {
        require(_timeInBlocks > 0, "Value cannot be less then 1");
        graceTime = _timeInBlocks;
    }

    function setMaxLendingPeriod(uint256 _maxLendingPeriod) public onlyAdmin {
        require(_maxLendingPeriod > 0, "Value cannot be less then 1");
        maxLendingPeriod = _maxLendingPeriod;
    }

    function addSupportedAsset(
        address veNFTAddress,
        address cheeTokenAddress,
        address tokenAddress
    ) public onlyAdmin {
        require(veNFTAddress != address(0) && cheeTokenAddress != address(0) && tokenAddress != address(0), "Incorrect Input");
        supportedAddress1[veNFTAddress] = cheeTokenAddress;
        supportedAddress2[veNFTAddress] = tokenAddress;
    }

    function changeDepositAddress(address _address) public onlyAdmin {
        require(_address != address(0), "Zero Address Detected");
        depositAddress = _address;
    }
```

Listing 3.4: `Lend.sol`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` 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 changed 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 confirmed.

4 Conclusion

In this audit, we have analyzed the `Chee Finance Lend` design and implementation. The `Chee Finance Lend` feature is designed for users to deposit `veNFTs`, and mint `cheeTokens` accordingly. 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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.