SMART CONTRACT AUDIT REPORT

for

CNHCTOKEN

Prepared By: Shuxiao Wang

Hangzhou, China

Oct. 12, 2020

Document Properties

Client	CNHC
Title	Smart Contract Audit Report
Target	CNHCToken
Version	1.0
Author	Huaguo Shi
Auditors	Huaguo Shi, Chiachih Wu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	Oct. 12, 2020	Huaguo Shi	Final Release
0.1	Oct. 09, 2020	Huaguo Shi	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the **CNHCToken** smart contract, we in the report outline our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract can be further improved due to the presence of some issues related to ERC20-compliance, security, or performance. This document outlines our audit results.

1.1 About CNHCToken

`CNHC` is a stablecoin pegged to offshore `Chinese Yuan`, and its smart contract implementation supports a number of features such as blacklist management, transaction fee setting, contract suspension and upgrades, etc. The basic information of CNHCToken is as follows:

Table 1.1: Basic Information of CNHCToken

Item	Description
Issuer	CNHC
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	Oct. 12, 2020

In the following, we show the list of reviewed contracts used in this audit:

- Contract Code: contracts.zip (md5: f0c14f886848ce2e016560195815b592)
- Contract Code: contracts.zip (md5: 4f9404e58e304622235cf3122c5cff4f)

1.2 About PeckShield

PeckShield Inc. [6] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [5]:

- 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 1.2: Vulnerability Severity Classification

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

We perform the audit according to the following procedures:

- 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.

- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

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
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
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 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.

1.4 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 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `CNHCToken` design and 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 ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	1	■
Informational	0	
Total	2	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions of each of them are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. Also, there is no critical or high severity issue, although the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 1 low-severity vulnerability.

Table 2.1: Key CNHCToken Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Limited Upgradeability of CNHCToken Contract	Business Practices	Confirmed
PVE-002	Medium	Explicit Owner Privileges in CNHCToken Contract	Security Features	Confirmed

Besides recommending specific measures 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 our detailed compliance checks and Section 4 for elaboration of reported issues.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Check Item	Description	Pass
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited CNHCToken. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted

ERC20 specification.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Check Item	Description	Pass
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approve() event	Is emitted on any successful call to approve()	✓

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	✓
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausible	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	✓
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

4 Detailed Results

4.1 Limited Upgradeability of CNHCToken Contract

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CNHCToken`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

In the `CNHCToken` contract, the `deprecate()` routine is designed as the key mechanism to upgrade `CNHCToken`. For illustration, we show below its implementation. In essence, it basically updates two internal variables: `deprecated` and `upgradedAddress`. The `deprecated` variable records the state of being upgraded while the `upgradedAddress` variable keeps a copy of the upgraded implementation. Note that the `deprecate` variable will redirect a number of calls to the upgraded version if the contract has been upgraded. The affected calls include `balanceOf()`, `totalSupply()`, `allowance()`, `transfer()`, `transferFrom()`, `approve()`, `increaseAllowance()`, and `decreaseAllowance()`.

```
152     // deprecate
153     function deprecate(address _upgradedAddress) public onlyOwner {
154         require(!deprecated, "CNHCToken: contract was deprecated");
155         require(_upgradedAddress != address(0));
156         deprecated = true;
157         upgradedAddress = _upgradedAddress;
158         emit Deprecate(_upgradedAddress);
159     }
```

Listing 4.1: CNHCToken.sol

In the following, we further elaborate one affected routine, i.e., `transfer()`. Note that if the `deprecate` variable is true, it would redirect the call to `transfer()`/`transferFrom()` of the upgraded contract. However, since the call is not a `delegate` call, a potential pitfall may exist: If the contract

has been upgraded, any call of `transfer()`/`transferFrom()` would be redirected to `transferByLegacy()`/`transferFromByLegacy()` in the upgraded contract, which eventually stores the balances of all holders and handles any possible transfers. This has an implicit assumption of fully migrating states from the old contract to the upgraded contract.

```
60      // normal functions
61      function transfer(address recipient, uint256 amount) public override isNotBlackUser(
            _msgSender()) returns (bool) {
62          require(!isBlackListUser(recipient), "BlackList: recipient address is in
                blacklist");
63
64          if (deprecated) {
65              return UpgradedStandardToken(upgradedAddress).transferByLegacy(_msgSender(),
                    recipient, amount);
66          } else {
67              return super.transfer(recipient, amount);
68          }
69      }
70
71      function transferFrom(address sender, address recipient, uint256 amount) public
            override isNotBlackUser(_msgSender()) returns (bool) {
72          require(!isBlackListUser(sender), "BlackList: sender address is in blacklist");
73          require(!isBlackListUser(recipient), "BlackList: recipient address is in
                blacklist");
74
75          if (deprecated) {
76              return UpgradedStandardToken(upgradedAddress).transferFromByLegacy(
                    _msgSender(), sender, recipient, amount);
77          } else {
78              return super.transferFrom(sender, recipient, amount);
79          }
80      }
```

Listing 4.2: CNHCToken.sol

Moreover, we notice that the current implementation can only be upgraded once. In other words, the second upgrade will not be possible. In fact, the second upgrade will be blocked because of the check at line 154: `require(!deprecated), "CNHCToken: contract was deprecated")`.

Recommendation Recommend the use of OpenZeppelin's `UpgradeabilityProxy` to better handle contract upgradeability.

Status This issue has been confirmed. Considering the possibility of actually upgrading the contract is extremely rare, the team decides to leave the current implementation as it is and will carefully and thoroughly check the update logic when such a need arises.

4.2 Explicit Owner Privileges in CNHCToken Contract

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: `CNHCToken, Votable`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In CNHCToken, the `_owner` account plays a critical role in governing and regulating the entire operation and maintenance (e.g., privileged contract management, account blacklisting, and minting/burning). It also has the privilege to pause the current token contract for withdrawals or upgrade it to implement additional functionalities or features.

For example, in `CNHCToken`, the `_owner` is the account who can access or execute all the privileged functions (marked with the `onlyOwner` modifier). While this may be normal or typically required for certain privileged operations, it could also be concerning as some of these privileged functions (only accessible to the `_owner` account) may be challenged from the community. For instance, the `openMintProposal()` function can be used to issue new tokens without being capped. 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 counterparty risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these `onlyOwner` privileges explicit or raising necessary awareness among contract users.

The `Votable` contract also shares a similar issue. Specifically, since the first voter is the same as the owner of the contact and adding other voters need to be voted by the first voter, the owner of the contract can essentially control the voting results.

Recommendation Make the list of extra privileges granted to `_owner` explicit to CNHCToken users. For the `Votable` owner, we instead recommend to solicit a list of trusted voters and publicly announce the results in the website (or the white-paper). By doing so, we can effectively accommodate possible concerns from the community.

Status This issue has been confirmed. Considering the need of a privileged account and the assumed trust of the `_owner` account, the team decided to leave it as is.

5 | Conclusion

In this audit, we have examined the `CNHCToken` design and implementation. `CNHCToken` is designed to enable organizations to protect their digital data against possible fraud and manipulation. We have accordingly checked all aspects related to ERC20 standard compatibility and known ERC20 pitfalls/vulnerabilities, and no issue was found in these areas. We have also proceeded to examine other areas such as coding practices and business logics. Overall, we found two issues of varying severities and these issues are promptly addressed by the team. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

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

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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