SMART CONTRACT AUDIT REPORT

for

HeartCoin (HTC)

Prepared By: Xiaomi Huang

PeckShield

January 26, 2024

Document Properties

Client	HeartGames
Title	Smart Contract Audit Report
Target	HTC
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 26, 2024	Xuxian Jiang	Final Release
1.0-rc	January 24, 2024	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 HeartCoin token contract (HTC), we outline in the report 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 is well-implemented without ERC20-compliance issues. The contract may still be improved by addressing the reported issues. This document outlines our audit results.

1.1 About HeartCoin

HeartCoin (HTC) strives to establish an integrated HeartGames Universe ecosystem where DeFi, GameFi , P2E, and the community operate in perfect synergy. In this ecosystem, NFTs and the HTC are meticulously balanced according to supply and demand, fostering a virtuous growth cycle. This audit covers the ERC20-compliance and security of its token contract. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of HTC

Item	Description
Name	HeartGames
Type	ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 26, 2024

In the following, we show the deployment address of the audited token contract:

- HTC: https://arbiscan.io/token/0xc0baa7cdf5b539d29a1d49fb230361507678b4d2

1.2 About PeckShield

PeckShield Inc. [8] 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 [7]:

- 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 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `HeartCoin (HTC)` token contract. 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	2	■ ■
Informational	0	
Total	3	

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 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. While there is no critical or high severity issue, 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 HTC Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Arithmetic Underflow in unlockByQuantity()	Coding Practices	Resolved
PVE-002	Low	Possible Locked Transfer to Frozen Accounts in transferWithLock()	Business Logic	Resolved
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate the above issue(s), 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

Item	Description	Status
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 `HTC` token contract. 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

Item	Description	Status
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	✓
Approval() 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, 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	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Whitelistable	The token contract allows the owner or privileged users to whitelist a specific address such that only token transfers and other operations related to that address are allowed	✓
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 Possible Arithmetic Underflow in unlockByQuantity()

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

Description

The HTC token contract has the built-in support of locking or unlocking tokens for certain accounts. While examining the unlocking logic, we notice the possibility of leading to an undesirable arithmetic underflow issue.

To elaborate, we show below the related `unlockByQuantity()` routine, which is used to unlock certain amount from the given holder. It has a rather straightforward logic in calculating total locked tokens, unlocking them, and next locking remaining amount. However, the inner `for`-loop may undesirably introduce an arithmetic underflow if the `idx` variable is equal to `0` when executing `idx -= 1` (line 682). Fortunately, the next statement of `idx++` (line 679) performs a reverse arithmetic overflow to get it corrected. Nevertheless, there is still a need to avoid unnecessary underflows and overflows.

```
666     function unlockByQuantity(address holder, uint256 value, uint256 releaseTime) public
             onlyPauser returns (bool) {
667         //1
668         require(!frozenAccount[holder]);
669         //2
670         require(timelockList[holder].length >0);
671 
672         //3
673         uint256 totalLocked;
674         for(uint idx = 0; idx < timelockList[holder].length ; idx++ ){
675             totalLocked = totalLocked.add(timelockList[holder][idx]._amount);
676         }
```

```
        require(totalLocked >value);

        //4
        for(uint idx = 0; idx < timelockList[holder].length ; idx++ ) {
            if( _unlock(holder, idx) ) {
                idx -=1;
            }
        }

        //5
        _lock(holder,totalLocked.sub(value),releaseTime);

        return true;
    }
```

Listing 4.1: `HTC::unlockByQuantity()`

Recommendation Revise the above routine to avoid the introduction of undesired arithmetic underflow and overflow. Note the same issue is also applicable to another routine `_autoUnlock()`.

Status The issue has been resolved as the underflow is intended.

4.2 Possible Locked Transfer to Frozen Accounts in transferWithLock()

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

Description

The `HTC` token contract allows the privileged pauser to freeze a specific account or lock tokens. And the token contract maintains an invariant in enforcing that a frozen account should not have any locked tokens. However, our analysis shows this invariant may be violated.

```
    function transferWithLock(address holder, uint256 value, uint256 releaseTime) public
         onlyPauser returns (bool) {
        _transfer(msg.sender, holder, value);
        _lock(holder,value,releaseTime);
        return true;
    }
```

Listing 4.2: `HTC::transferWithLock()`

To elaborate, we show above the related `transferWithLock()` routine. This routine allows to transfer tokens to a recipient and the transferred tokens will be immediately locked until the given `releaseTime` is expired. However, it does not validate whether the recipient is a frozen account or not. To maintain the above invariant, there is a need to add the following statement, i.e., `require(!frozenAccount[holder]);`.

Recommendation Validate the recipient is not frozen in the above `transferWithLock()` routine.

Status The issue has been resolved as the team confirms it is only used when distributing tokens within the team, not to general users such as exchanges.

4.3 Trust Issue of Admin Keys

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

Description

In the `HTC` token contract, there is a privileged admin account `owner` that plays a critical role in regulating the token-wide operations (e.g., pause the contract and assign pauser role). In the following, we show the representative function potentially affected by this privilege.

```
    function freezeAccount(address holder) public onlyPauser returns (bool) {
        require(!frozenAccount[holder]);
        require(timelockList[holder].length == 0);
        frozenAccount[holder] = true;
        emit Freeze(holder);
        return true;
    }

    function unfreezeAccount(address holder) public onlyPauser returns (bool) {
        require(frozenAccount[holder]);
        frozenAccount[holder] = false;
        emit Unfreeze(holder);
        return true;
    }

    function lockByQuantity(address holder, uint256 value, uint256 releaseTime) public
        onlyPauser returns (bool) {
        require(!frozenAccount[holder]);
        _lock(holder,value,releaseTime);
        return true;
    }
```

```
    function unlockByQuantity(address holder, uint256 value, uint256 releaseTime) public
         onlyPauser returns (bool) {
        //1
        require(!frozenAccount[holder]);
        //2
        require(timelockList[holder].length >0);

        //3
        uint256 totalLocked;
        for(uint idx = 0; idx < timelockList[holder].length ; idx++ ){
            totalLocked = totalLocked.add(timelockList[holder][idx]._amount);
        }
        require(totalLocked >value);

        //4
        for(uint idx = 0; idx < timelockList[holder].length ; idx++ ) {
            if( _unlock(holder, idx) ) {
                idx -=1;
            }
        }

        //5
        _lock(holder,totalLocked.sub(value),releaseTime);

        return true;
    }

    function transferWithLock(address holder, uint256 value, uint256 releaseTime) public
         onlyPauser returns (bool) {
        _transfer(msg.sender, holder, value);
        _lock(holder,value,releaseTime);
        return true;
    }
```

Listing 4.3: An Example Privileged Operation in HTC

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it would be worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the new owner to modify a number of sensitive system parameters, which may directly undermine the assumption of the token design.

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 The issue has been mitigated with the use of multi-sig to manage the admin key.

5 Conclusion

In this security audit, we have examined the HeartCoin (HTC) token design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, there are no critical level vulnerabilities discovered and other identified issues are promptly addressed.

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

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