SMART CONTRACT AUDIT REPORT

for

LuckyChip Token

Prepared By: Yiqun Chen

PeckShield

February 17, 2022

Document Properties

Client	LuckyChip
Title	Smart Contract Audit Report
Target	LuckyChip
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	February 17, 2022	Yiqun Chen	Final Release
1.0-rc	February 17, 2022	Yiqun Chen	Release Candidate

Contact

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

Name	Yiqun Chen
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 **LuckyChip** token contract, 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 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 LuckyChip

LuckyChip is an ERC20-compliant token developed using the excellent smart contract bases from `OpenZeppelin`. The main features of the `LuckyChip` token include the full ERC20 compatibility and the voting power of the `LuckyChip` token holders and their delegates.

The basic information of LuckyChip is as follows:

Table 1.1: Basic Information of LuckyChip

Item	Description
Issuer	LuckyChip
Website	https://luckychip.io/
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	February 17, 2022

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

- https://github.com/luckychip-io/core/blob/master/contracts/token/LCToken.sol (0d957aa)

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

- https://github.com/luckychip-io/core/blob/master/contracts/token/LCToken.sol (479d584)

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 `LuckyChip` 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	1	■
Medium	1	■
Low	0	
Informational	1	■
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. However, the smart contract implementation can be improved because of the existence of 1 high-severity vulnerability, 1 medium-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key LuckyChip Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Voting Amplification With Sybil Attacks	Business Logics	Fixed
PVE-002	Medium	Trust Issue Of Admin Roles	Security Features	Mitigated
PVE-003	Informational	Consistency Between Function Definitions And Return Statements	Coding Practices	Fixed

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 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 LuckyChip. 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-

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()	✓

adopted ERC20 specification. 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/ERC2222), 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	—
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 Voting Amplification With Sybil Attacks

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: LCToken
- Category: Business Logics [6]
- CWE subcategory: CWE-841 [3]

Description

The LuckyChip tokens can be used for governance in allowing for users to cast and record the votes. Moreover, the LCToken contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the number of votes are counted via getPriorVotes().

Our analysis shows that the current governance functionality is vulnerable to a new type of so-called Sybil attacks. For elaboration, let's assume at the very beginning there is a malicious actor named Malice, who owns 100 LuckyChip tokens. Malice has an accomplice named Trudy who currently has 0 balance of LuckyChip tokens. This Sybil attack can be launched as follows:

```
    function _delegate(address delegator, address delegatee)
        internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying FATs (not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
```

```
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].
                    votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].
                    votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }
```

Listing 4.1: `LCToken.sol`

1. `Malice` initially delegates the voting to `Trudy`. Right after the initial delegation, `Trudy` can have 100 votes if he chooses to cast the vote.

2. `Malice` transfers the full 100 balance to M_1 who also delegates the voting to `Trudy`. Right after this delegation, `Trudy` can have 200 votes if he chooses to cast the vote. The reason is that the `LuckyChip` contract's `transfer()` does NOT `_moveDelegates()` together. In other words, even now `Malice` has 0 balance, the initial delegation (of `Malice`) to `Trudy` will not be affected, therefore `Trudy` still retains the voting power of 100 `LuckyChip` tokens. When M_1 delegates to `Trudy`, since M_1 now has 100 `LuckyChip` tokens, `Trudy` will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 `LuckyChip` balance to M_{i+1} who also delegates the votes to `Trudy`. Every iteration will essentially add 100 voting power to `Trudy`. In other words, we can effectively amplify the voting powers of `Trudy` arbitrarily with new accounts created and iterated!

Recommendation To mitigate, it is necessary to accompany every single `transfer()` and `transferFrom()` with the `_moveDelegates()` so that the voting power of the sender's delegate will be moved to the destination's delegate. By doing so, we can effectively mitigate the above `Sybil` attacks. Since the contract is already deployed, it is safe and acceptable to deploy another contract for governance, and use the current one for other ERC-20 functions only. A cleaner solution would be to migrate the current contract to a new one with the suggested fix, but the migration effort may be costly.

Status This issue has been fixed in this commit: `ba58987`.

4.2 Trust Issue Of Admin Roles

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `LCToken`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `LCToken` token contract, there is a privileged `owner` account (assigned in the `constructor`) that plays a critical role in governing and regulating the token-related operations (e.g., add/remove minters).

To elaborate, we show below the `mint()/burn()/addMinter()/removeMinter()` functions in the `LCToken`contract. The `mint()` function allows the `minter` to add unrestricted tokens into circulation and the recipient can be directly provided when the mint operation takes place.

```
    function addMinter(address _addMinter) public onlyOwner returns (bool) {
        require(_addMinter != address(0), "Token: _addMinter is the zero address");
        return EnumerableSet.add(_minters, _addMinter);
    }

    function delMinter(address _delMinter) public onlyOwner returns (bool) {
        require(_delMinter != address(0), "Token: _delMinter is the zero address");
        return EnumerableSet.remove(_minters, _delMinter);
    }
```

Listing 4.2: `LCToken::addMinter()/delMinter()`

```
    // @notice Creates `_amount` token to `_to`.
    function mint(address _to, uint256 _amount) public onlyMinter {
        _mint(_to, _amount);
        _moveDelegates(address(0), _delegates[_to], _amount);
    }

    // @notice burn `_amount` token from `_from`.
    function burn(address _from, uint256 _amount) public onlyMinter {
        _burn(_from, _amount);
        _moveDelegates(_delegates[_from], address(0), _amount);
    }
```

Listing 4.3: `LCToken::mint()/burn()`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the admin roles 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 contract users.

Recommendation Make the list of extra privileges granted to `owner` explicit to LuckyChip users.

Status This issue has been mitigated in this commit: `479d584`.

4.3 Consistency Between Function Definitions And Return Statements

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `LCToken`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `LCToken` contract, the `delegate()/delegateBySig()` functions are used to delegate votes from `msg.sender` or an intended signatory to the given `delegatee`. As shown (line 108) in the following code snippet, the `delegate()` function returns by calling the internal routine `_delegate()`, and the `return` keyword is not needed as the function is declared without any return value.

```
107      function delegate(address delegatee) external {
108        return _delegate(msg.sender, delegatee);
109    }
110
111    /**
112     * @notice Delegates votes from signatory to `delegatee`
113     * @param delegatee The address to delegate votes to
114     * @param nonce The contract state required to match the signature
115     * @param expiry The time at which to expire the signature
116     * @param v The recovery byte of the signature
117     * @param r Half of the ECDSA signature pair
118     * @param s Half of the ECDSA signature pair
119     */
120    function delegateBySig(
121        address delegatee,
122        uint nonce,
123        uint expiry,
124        uint8 v,
125        bytes32 r,
```

```
        bytes32 s
    )
        external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "LC::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "LC::delegateBySig: invalid nonce");
        require(now <= expiry, "LC::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }
```

Listing 4.4: LCToken.sol

The same issue is also applicable to the `delegateBySig()` function.

Recommendation Remove the `return` keyword in the above two functions. An example revision is shown as follows.

```
    function delegate(address delegatee) external {
        _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to 'delegatee'
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
```

```
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "LC::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "LC::delegateBySig: invalid nonce");
        require(now <= expiry, "LC::delegateBySig: signature expired");
        _delegate(signatory, delegatee);
    }
```

Listing 4.5: LCToken.sol (revised)

Status This issue has been fixed in this commit: `ccf8a3c`.

5 | Conclusion

In this security audit, we have examined the design and implementation of the `LuckyChip` token contract. 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, although no critical or high level vulnerabilities were discovered, we identified two issues that were promptly confirmed and addressed by the team. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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.