SMART CONTRACT AUDIT REPORT

for

Coin98 Staking

Prepared By: Yiqun Chen

PeckShield
December 8, 2021

Document Properties

Client	Coin98
Title	Smart Contract Audit Report
Target	Coin98 Staking
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(s)	Description
1.0	December 8, 2021	Yiqun Chen	Final Release
1.0-rc	November 18, 2021	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 4
  1.1 About Coin98 Staking . . . 4
  1.2 About PeckShield . . . 5
  1.3 Methodology . . . 5
  1.4 Disclaimer . . . 9
2 Findings 10
  2.1 Summary . . . 10
  2.2 Key Findings . . . 11
3 Detailed Results 12
  3.1 Trust Issue of Admin Keys . . . 12
  3.2 Improper Update Of totalStaked . . . 13
  3.3 Improper maxStaked Enforcement in stake() . . . 14
  3.4 Improved Sanity Checks For System Parameters . . . 16
  3.5 Potential Less Profit From Permissionless unstake() . . . 17
4 Conclusion 19
References 20

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Coin98 Staking 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 Coin98 Staking

In the Coin98 Staking protocol, the user can receive the non-fungible token (NFT) as a certificate by staking certain amount of C98 tokens into the contract. There are two default system parameters set by the owner account, locked_time and floating_rate. Users have to stake at least locked_time (e.g., 15 days) to redeem, and get rewards calculated with floating_rate. There are also many different packages registered by the owner account, and each has its own staking rules for users. Once users stake over the specified time, they will enjoy higher reward rate.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Coin98 Staking

Item	Description
Name	Coin98
Website	https://coin98.com/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 8, 2021

In the following, we show the contract file and the MD5/SHA checksum value of the contract file:

- File: C98Stake.sol
- MD5: 22f63f87847c85c93c95b888e4696c2b
- SHA: 7bb10b8eb2ecfbd27f5abcb07bf8e961dd15b6390b4a733ea8f09bfa04a29b93

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

- https://github.com/coin98/coin98-stake.git (a596d3e)

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 accordingly classified into four categories, 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 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

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.

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 design and implementation of the `Coin98 Staking` 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	2	■ ■
Low	3	■ ■ ■
Informational	0	
Total	5	

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 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 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-002	Low	Improper Update Of totalStaked	Coding Practices	Fixed
PVE-003	Low	Improper maxStaked Enforcement in stake()	Coding Practices	Fixed
PVE-004	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-005	Medium	Potential Less Profit From Permissionless unstake()	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Coin98Stake
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Coin98 Staking protocol, there is a special administrative account, i.e., owner. This owner account plays a critical role in governing and regulating the system-wide operations (e.g., system parameter setting). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and its related privileged access in current contract.

To elaborate, we show below the withdraw() function in the Coin98Stake contract. This function allows the owner to withdraw all the C98Token staked in the contract.

```
    function withdraw(uint256 _amount) public onlyOwner {
        require(_amount > 0);
        require(C98Token.balanceOf(address(this)) >= _amount);
        C98Token.transfer(msg.sender, _amount);
    }
```

Listing 3.1: Coin98Stake::withdraw()

Note that it could be worrisome if the privileged owner account is a plain EOA account. The discussion with the team confirms that the owner account is currently managed by a multi-sig account. However, 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.

3.2 Improper Update Of totalStaked

- ID: PVE-002
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Coin98Stake`
- Category: Coding Practices [3]
- CWE subcategory: CWE-841 [4]

Description

In the `Coin98Stake` contract, there is a public variable `totalStaked`, which is used to record the total amount of `C98` tokens staked in the contract. The `Coin98Stake` contract also provides an `unstake()` function for users to redeem their tokens, and get their rewards. If users unstake successfully, the `totalStaked` should be decreased accordingly. However, it comes to our attention that the `totalStaked` is not updated properly.

To elaborate, we show below the code snippet of the `unstake()` routine. After the transfer of `C98` tokens (line 1629), this routine decreases the `totalStaked` with the unstake amount (line 1631), but doesn't update it. As a result, the `totalStaked` state is not updated. The same issue is also applicable to the `stake()` routine in the `Coin98Stake` contract.

```
1619     function unstake(uint256 _tokenId) public {
1620         StakeInfo storage stakeInfo = StakeInfos[_tokenId];
1621         uint256 _profit = getStakedByTokenId(_tokenId);
1622         require(_profit > 0, "Not meet unstake condition");
1623         require(ownerOf(_tokenId) == stakeInfo.owner, "Not meet owner condition");
1624 
1625         uint256 _profitTotal = _profit.add(stakeInfo.amount);
1626 
1627         require(C98Token.balanceOf(address(this)) >= _profitTotal);
1628         stakeInfo.flag = false;
1629         C98Token.transfer(stakeInfo.owner, _profitTotal);
1630 
1631         totalStaked.sub(stakeInfo.amount);
1632         emit _unstake(_tokenId, _profitTotal, stakeInfo.time);
1633     }
```

Listing 3.2: `Coin98Stake::unstake()`

Recommendation Change the statement of `totalStaked.sub(stakeInfo.amount)` to `totalStaked = totalStaked.sub(stakeInfo.amount)` in the `unstake()` function. And change the statement of `totalStaked.add(_amount)` to `totalStaked = totalStaked.add(_amount)` in the `stake()` function.

Status This issue has been fixed as suggested.

3.3 Improper maxStaked Enforcement in stake()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Coin98Stake`
- Category: Coding Practices [3]
- CWE subcategory: CWE-841 [4]

Description

As described in Section 3.2, the public variable `totalStaked` is used for recording the total amount of `C98` tokens staked in the contract. And it can not exceed the `maxStaked`, another public variable which is set to 1000000000 ether in default. There is a check in the `stake()` function to ensure the `totalStaked` is less than `maxStaked`. However, it fails to enforce `maxStaked`.

```
    function stake(uint256 _amount, string memory _name, string memory _package,uint256 _customID) public {
        require(validPackage(_package), "Package not found");

        PackageInfo memory pkInfo = PackageInfos[_package];
        // Check the validity of the package min, max & the amount of transferFrom
        require(_amount > 0 && _amount >= pkInfo.min && _amount < pkInfo.max , "Wrong min max format");

        require(totalStaked <= maxStaked, "Maximum number of staked");

        bool _isCustomID = _customID != 0;
        //Validate the custom name if basing on C98 Ref ID Rule, it will be free of change
        uint256 nameSize = bytes(_name).length;
        bool _isNotCustomname = Convertible.compareStrings(Convertible.sliceString(1,3, _name), ref_id) && nameSize == 10;

        if(!_isNotCustomname){
            require(nameSize <= 20 && nameSize > 0,"Not meet name condition");
        }

        require(C98Token.transferFrom(msg.sender, address(this), _amount.add(_isCustomID ? id_fee : 0 ).add(_isNotCustomname? 0 : naming_fee)));


```

```
1554         string memory randomID;
1555
1556         if (_isCustomID){
1557             randomID = Convertible.uint2str(_customID);
1558         } else {
1559             uint256 total = totalToken();
1560             string memory randomConvert = Convertible.uint2str(uint256(keccak256(abi.
                 encodePacked(total.add(1),
1561                 _amount,block.timestamp,nft_prefix))));
1562             randomID = Convertible.sliceString(10,21,randomConvert);
1563         }
1564
1565         // Random string after prefix is fixed at 12
1566         require(bytes(randomID).length == 12);
1567
1568         // Token ID start with nft_prefix
1569         uint256 nftPackageId = Convertible.bytesToUInt(Convertible.stringToBytes32(
             string(abi.encodePacked(nft_prefix,randomID))));
1570
1571         require(!_exists(nftPackageId), "ERC721: token already minted");
1572
1573         // Storage stake information
1574         StakeInfo storage stakeInfo = StakeInfos[nftPackageId];
1575         stakeInfo.flag = true;
1576         stakeInfo.owner = msg.sender;
1577         stakeInfo.amount = _amount;
1578         stakeInfo.time = block.timestamp;
1579         stakeInfo.packgeId = _package;
1580
1581
1582         stakeInfo.name = _name;
1583         stakeInfo.isCustomID = _isCustomID;
1584
1585         stakeInfo.packageTime = pkInfo.time;
1586         stakeInfo.rate = pkInfo.rate;
1587
1588         totalStaked.add(_amount);
1589         _mintAnElement(msg.sender, nftPackageId, _isCustomID);
1590     }
```

Listing 3.3: `Coin98Stake::stake()`

Specifically, we show above the code snippet of the `stake()` routine. Note that the check of the `totalStaked` (line 1540) is before the update of the `totalStaked` (line 1588), so it's possible for the `totalStaked` to exceed the `maxStaked` after passing the check.

Recommendation Move the statement of `totalStaked = totalStaked.add(_amount)` before the check of `totalStaked` (`require(totalStaked <= maxStaked, "Maximum number of staked")`).

Status This issue has been fixed as suggested.

3.4 Improved Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Coin98Stake
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Coin98 Staking` protocol is no exception. Specifically, if we examine the `Coin98Stake` contract, their is a `PackageInfo` struct, and it has defined the following parameters, e.g., `min`, `max`, `time`, and `rate`. These parameters define the minimum staking amount, maximum staking amount, required staking time to enjoy the higher reward rate, and the reward rate, respectively.

In the following, we show he corresponding routines that allow for their changes.

```
    function configurePackage(
        string memory _package,
        uint256 _min,
        uint256 _max,
        uint256 _time,
        uint256 _rate
    )
        public
        onlyOwner()
    {
        require(validPackage(_package), "Package not found");
        require(_min>0 && _max >0 && _min < _max, "Wrong numeric format");

        PackageInfos[_package].min = _min;
        PackageInfos[_package].max = _max;
        PackageInfos[_package].time = _time;
        PackageInfos[_package].rate = _rate;
    }

    function register(
        string memory _package,
        uint256 _min,
        uint256 _max,
        uint256 _time,
        uint256 _rate
    )
        public
        onlyOwner()
    {
        require(!validPackage(_package), "Package already existed");
        require(_min>0 && _max >0 && _min < _max , "Wrong numeric format");
```

```
        PackageInfos[_package].min = _min;
        PackageInfos[_package].max = _max;
        PackageInfos[_package].time = _time;
        PackageInfos[_package].rate = _rate;
    }
```

Listing 3.4: `Coin98Stake::configurePackage()/register()`

Our result 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 a small `rate` (less than `floating_rate`) and a large `time` (larger than `locked_time`) will lead to the result of less profits but longer staking time.

Recommendation Add the statement of `_time > locked_time && _rate > floating_rate` in the `configurePackage()` function and the `register()` function.

Status This issue have been fixed as suggested.

3.5 Potential Less Profit From Permissionless unstake()

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Coin98Stake`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.2, the `Coin98Stake` contract provides an `unstake()` function for users to redeem their tokens, and get their rewards. However, users are not able to redeem until the staking time exceeds the `locked_time`. They can also choose to continue the staking to gain higher rewards with higher reward rate.

However, we find that the `unstake()` function is permissionless, which can be invoked by anyone. In the following, we list below the related `unstake()` function.

```
    function unstake(uint256 _tokenId) public {
        StakeInfo storage stakeInfo = StakeInfos[_tokenId];
        uint256 _profit = getStakedByTokenId(_tokenId);
        require(_profit > 0, "Not meet unstake condition");
        require(ownerOf(_tokenId) == stakeInfo.owner, "Not meet owner condition");

        uint256 _profitTotal = _profit.add(stakeInfo.amount);

```

```
1627         require(C98Token.balanceOf(address(this)) >= _profitTotal);
1628         stakeInfo.flag = false;
1629         C98Token.transfer(stakeInfo.owner, _profitTotal);
1630 
1631         totalStaked.sub(stakeInfo.amount);
1632         emit _unstake(_tokenId, _profitTotal, stakeInfo.time);
1633     }
```

Listing 3.5: `Coin98Stake::unstake()`

In the `unstake()` function, there is a `require` statement (line 1623), which checks if the owner of the `NFT` is original. However, there is no check for `msg.sender`, which means everyone can call the `unstake()` function to redeem for others. As a result, the user will gain less profits if the redeem is brought forward.

Recommendation Replace the statement of `require(ownerOf(_tokenId)== stakeInfo.owner, "Not meet owner condition")` with `require(ownerOf(_tokenId)== msg.sender, "Not meet owner condition")`.

Status This issue have been fixed as suggested.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Coin98 Staking` protocol. The `Coin98 Staking` protocol mints different `NFTs` with different staking rules as certificates for users who stake their `C98` tokens into the contract. 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-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-758: Reliance on Undefined, Unspecified, or Implementation-Defined Behavior. https://cwe.mitre.org/data/definitions/758.html.

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

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

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.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.