



# SMART CONTRACT AUDIT REPORT

for

Coin98Multisig



Prepared By: Yiqun Chen

PeckShield  
August 09, 2021

## Document Properties

Client	Coin98 Labs LTD
Title	Smart Contract Audit Report
Target	Coin98Multisig
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

## Version Info

Version	Date	Author(s)	Description
1.0	August 09, 2021	Xuxian Jiang	Final Release
1.0-rc2	August 09, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	August 07, 2021	Xuxian Jiang	Release Candidate #1

## 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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	About Coin98Multisig . . . . .	4
1.2	About PeckShield . . . . .	5
1.3	Methodology . . . . .	5
1.4	Disclaimer . . . . .	6
<b>2</b>	<b>Findings</b>	<b>10</b>
2.1	Summary . . . . .	10
2.2	Key Findings . . . . .	11
<b>3</b>	<b>Detailed Results</b>	<b>12</b>
3.1	Improved Validation Of Function Arguments . . . . .	12
3.2	Potential Reentrancy Risk in Coin98Multisig::vote() . . . . .	13
3.3	Redundant State/Code Removal . . . . .	14
<b>4</b>	<b>Conclusion</b>	<b>16</b>
	<b>References</b>	<b>17</b>

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the Coin98Multisig 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 Coin98Multisig

Coin98Multisig is a multi signature wallet on Ethereum and Binance Smart Chain (BSC). It allows the user to create a multisig wallet, configure the owners of the wallet with their respective voting weight and set the number of votes needed to perform the request. Each owner could create a request to vote and all owners are eligible to vote the request with their voting power. The request will be performed if the number of votes exceeds the configured threshold.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Coin98Multisig

Item	Description
Target	Coin98Multisig
Website	<a href="https://www.coin98.com">https://www.coin98.com</a>
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 09, 2021

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

- <https://etherscan.io/address/0x0ece57a677d5e72d1ad45774239e23463cf1d743#code>

- <https://bscscan.com/address/0x82c3da62b7db06e93c67ba90330ccedeef725a63#code>

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

- <https://github.com/coin98/coin98-multisig.git> (10708da)

And here is the list of new contracts that have been deployed after fixing issues reported here:

- <https://etherscan.io/address/0x9ae5c1cf82af51cbb83d9a7b1c52af4b48e0bb5e#code>
- <https://bscscan.com/address/0x836bf46520c373fdc4dc7e5a3bae735d13bd44e3#code>

## 1.2 About PeckShield

PeckShield Inc. [7] 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](mailto:contact@peckshield.com)).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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) [5], 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.

## 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

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
<b>Configuration</b>	Weaknesses in this category are typically introduced during the configuration of the software.
<b>Data Processing Issues</b>	Weaknesses in this category are typically found in functionality that processes data.
<b>Numeric Errors</b>	Weaknesses in this category are related to improper calculation or conversion of numbers.
<b>Security Features</b>	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
<b>Time and State</b>	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.
<b>Error Conditions, Return Values, Status Codes</b>	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.
<b>Resource Management</b>	Weaknesses in this category are related to improper management of system resources.
<b>Behavioral Issues</b>	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
<b>Business Logics</b>	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.
<b>Initialization and Cleanup</b>	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
<b>Arguments and Parameters</b>	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
<b>Expression Issues</b>	Weaknesses in this category are related to incorrectly written expressions within code.
<b>Coding Practices</b>	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.





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 Coin98Multisig 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	1	
Low	0	
Informational	2	
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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 1 medium-severity vulnerability and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Improved Validation of Function Arguments	Coding Practices	Fixed
PVE-002	Medium	Potential Reentrancy Risk in Coin98Multisig::vote()	Time and State	Fixed
PVE-003	Informational	Redundant State/Code Removal	Coding Practices	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 Improved Validation Of Function Arguments

- ID: PVE-001
- Severity: Informational
- Likelihood: Low
- Impact: Low
- Target: Coin98Multisig
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

#### Description

In the Coin98Multisig contract, there is a public function `changeOwners()` that is designed to add/remove/change owners, with their respective voting weight. To elaborate, we show below the related code snippet of the function.

```

229
230     function changeOwners(address[] memory nOwners, uint16[] memory vPowers, uint16
        vRate)
231         selfOnly()
232         external override returns (bool) {
233             _changeOwners(nOwners, vPowers, vRate);
234             return true;
235         }
236
237     function _changeOwners(address[] memory nOwners, uint16[] memory vPowers, uint16
        vRate) internal {
238         VoteRequirement memory requirement = _voteRequirement;
239         uint256 i;
240         for (i = 0; i < nOwners.length; i++) {
241             address nOwner = nOwners[i];
242             uint16 cPower = _votePowers[nOwner];
243             uint16 vPower = vPowers[i];
244             ...
245         }
246     }
247 }
```

Listing 3.1: Coin98Multisig::`changeOwners()`

It comes to our attention that the `changeOwners()` function has the inherent assumption that the length of the given array, `nOwners`, would be equal to the length of another given array, `vPowers`. However, this is not enforced inside the `_changeOwners()` function. If the length of these two given arrays are different, the execution of the `_changeOwners()` function could introduce unexpected result. We suggest to properly validate the given arrays to have the same length.

**Recommendation** Validate the given arguments of `changeOwners()` to have the same length.

**Status** The issue has been fixed by this commit: `0d51e58`.

## 3.2 Potential Reentrancy Risk in `Coin98Multisig::vote()`

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Coin98Multisig`
- Category: Time and State [4]
- CWE subcategory: CWE-663 [2]

### Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [9] exploit, and the recent `Uniswap/Lendf.Me` hack [8].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `Coin98Multisig` as an example, the `vote()` function (see the code snippet below) is provided to externally call a contract to execute the request if enough votes. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 189) starts before effecting the update on internal states (lines 191 and 192), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `vote()` function.

```

176     function vote()
177     isOwner(msg.sender)
178     public override returns (bool) {
179         VoteProgress memory progress = _voteProgress;
180         require(progress.requestId > 0, "Coin98Multisig: No pending request");

```

```

181     if (_votes[msg.sender] < progress.requestId) {
182         _votes[msg.sender] = progress.requestId;
183         progress.currentVote += _votePowers[msg.sender];
184         _voteProgress = progress;
185         emit Voted(msg.sender, progress.requestId, progress.currentVote, progress.
            requiredVote);
186     }
187     if (progress.currentVote >= progress.requiredVote) {
188         Request memory req = _request;
189         (bool success,) = req.destination.call{value: req.value}(req.data);
190         if (success) {
191             delete _request;
192             delete _voteProgress;
193             Executed(true, progress.requestId, req.destination, req.value, req.data)
                ;
194         }
195         else {
196             Executed(false, progress.requestId, req.destination, req.value, req.data
                );
197         }
198     }
199     return true;
200 }

```

Listing 3.2: Coin98Multisig :: vote()

**Recommendation** Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

**Status** The issue has been fixed by this commit: 10708da.

### 3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Coin98Multisig
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

#### Description

In the Coin98Multisig contract, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed. For example, the modifier `notOwner()` and the modifier `validVotingPower()` are defined but not used throughout the entire Coin98Multisig contract. If there is no usage of these modifiers, we suggest to simplify the contract by removing the redundant code.

```
18     modifier notOwner(address owner) {
19         require(_votePowers[owner] == 0, "Coin98Multisig: Already an owner");
20         _;
21     }

23     modifier validVotingPower(uint256 vPower) {
24         require(vPower > 0, "Coin98Multisig: Invalid vote weight");
25         _;
26     }
```

Listing 3.3: notOwner()/validVotingPower()

**Recommendation** Consider the removal of the redundant code with a simplified implementation.

**Status** The issue has been fixed by this commit: [0d51e58](#).



## 4 | Conclusion

In this audit, we have analyzed the `Coin98Multisig`'s design and implementation. `Coin98Multisig` is a multi signature wallet on Ethereum and Binance Smart Chain (BSC). It allows the user to create a multisig wallet which provides a multi signature platform for owners to vote the request could be performed. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

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-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [3] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [4] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [5] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [6] OWASP. Risk Rating Methodology. [https://www.owasp.org/index.php/OWASP\\_Risk\\_Rating\\_Methodology](https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology).
- [7] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [8] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [9] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.