SMART CONTRACT AUDIT REPORT

for

Creative Crowdfunding Protocol

Prepared By: Xiaomi Huang

PeckShield

May 20, 2025

Document Properties

Client	Creative Crowdfunding
Title	Smart Contract Audit Report
Target	CC Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 20, 2025	Xuxian Jiang	Final Release
1.0-rc	April 28, 2025	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 `Creative Crowdfunding` 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 CC Protocol

`Creative Crowdfunding (CC)` protocol is a decentralized crowdfunding protocol designed to help creators launch and manage campaigns across multiple platforms. By providing a standardized infrastructure, the protocol simplifies the process of creating, funding, and managing crowdfunding initiatives in `Web3` across different platforms. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of CC Protocol

Item	Description
Name	Creative Crowdfunding
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 20, 2025

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

- https://github.com/ccprotocol/ccprotocol-contracts.git (be34f00)

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

- https://github.com/ccprotocol/ccprotocol-contracts.git (a0a168e)

1.2 About PeckShield

PeckShield Inc. [10] 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	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- 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

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

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) [8], 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 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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the CC 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	4	■ ■ ■ ■
Low	1	■
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 4 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect updateGoalAmount() Logic in CampaignInfo	Business Logic	Resolved
PVE-002	Medium	Possibly Unauthorized Pledge in AllOrNothing	Business Logic	Resolved
PVE-003	Low	Timely Fee Update in CampaignInfo::updateSelectedPlatform()	Coding Practices	Resolved
PVE-004	Medium	Improved Validation of Function Arguments	Coding Practices	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Incorrect updateGoalAmount() Logic in CampaignInfo

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: CampaignInfo
- Category: Business Logic [7]
- CWE subcategory: CWE-770 [3]

Description

For each campaign, the CC protocol instantiates a standalone CampaignInfo contract from which the campaign owner may manage or update a variety of campaign-wide parameters. In the process of examining the logic to update the campaign goal amount, we notice current implementation is flawed.

In the following, we show the implementation of the related routine, i.e., updateGoalAmount(). It has a rather straightforward logic in updating the campaign's goal amount. However, current implementation only validates the given goal amount is not zero, but forgets to apply the new goal amount in the contract.

```
    function updateGoalAmount(
        uint256 goalAmount
    )
        external
        override
        onlyOwner
        currentTimeIsLess(getLaunchTime())
        whenNotPaused
        whenNotCancelled
    {
        if (goalAmount == 0) {
            revert CampaignInfoInvalidInput();
        }
        emit CampaignInfoGoalAmountUpdated(goalAmount);
    }
```

Listing 3.1: CampaignInfo::updateGoalAmount()

Recommendation Improve the above-mentioned routine to properly update the campaign's goal amount.

Status This issue has been fixed in the following PR: `13`.

3.2 Possibly Unauthorized Pledge in AllOrNothing

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AllOrNothing`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

Description

Besides instantiating a standalone a `CampaignInfo` for each campaign, the `CC` protocol allows to create the campaign-specific treasury contract so that an interested user may choose to pledge for the campaign. While examining the related pledge logic, we notice an issue where an user may be unknowingly pledging for a campaign.

In the following, we show the implementation of the related `pledgeWithoutAReward()` routine. Specifically, this routine validates the given input and adds the `backer` to pledge for the campaign. However, it comes to our attention that the `backer` may have allowed the token spending to the treasury contract, but does not authorize the calling user to actually pledge for the campaign. Note the pledge may lead to a new minted `NFT` to the `backer`, who may eventually be able to request a refund. But still, the unauthorized pledge is not intended. A proper fix requires the authorization from the `backer` to the calling user.

```
    function pledgeWithoutAReward(
        address backer,
        uint256 pledgeAmount
    )
        external
        currentTimeIsWithinRange(INFO.getLaunchTime(), INFO.getDeadline())
        whenCampaignNotPaused
        whenNotPaused
        whenCampaignNotCancelled
        whenNotCancelled
    {
        uint256 tokenId = s_tokenIdCounter.current();
        bytes32[] memory emptyByteArray = new bytes32[](0);

        _pledge(backer, ZERO_BYTES, pledgeAmount, 0, tokenId, emptyByteArray);
    }
```

Listing 3.2: `AllOrNothing::pledgeWithoutAReward()`

Recommendation Revise the above `pledgeWithoutAReward()` routine to properly validate the prior authorization from the given `backer` to the calling user. Note another `pledgeForAReward()` routine can be similarly improved.

Status This issue has been resolved as the team considers it as a design choice.

3.3 Timely Fee Update in CampaignInfo::updateSelectedPlatform()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `CampaignInfo`
- Category: Business Logic [7]
- CWE subcategory: N/A

Description

As mentioned earlier, the `CC` protocol instantiates a standalone `CampaignInfo` contract for each campaign. Through the `CampaignInfo` contract, the owner may be able to adjust the campaign-wide parameters. In the process of examining a specific setter to add a new selected platform, we notice current implementation needs to timely update the respective platform fee.

In the following, we show the implementation of the related routine, i.e., `updateSelectedPlatform()` . While it has properly validated the given `platformHash` is whitelisted in the protocol, when a selected platform is updated, there is a need to update the related platform fee as well. In other words, when `selection == True`, we need to add the following statement, `s_platformFeePercent[platformHash] = GLOBAL_PARAMS.getPlatformFeePercent(platformHash);` (right after line 411). Otherwise, we need to reset the platform fee.

```
394     function updateSelectedPlatform(
395         bytes32 platformHash,
396         bool selection
397     )
398         external
399         override
400         onlyOwner
401         currentTimeIsLess(getLaunchTime())
402         whenNotPaused
403         whenNotCancelled
404     {
405         if (checkIfPlatformSelected(platformHash) == selection) {
406             revert CampaignInfoInvalidInput();
407         }
408         if (!GLOBAL_PARAMS.checkIfPlatformIsListed(platformHash)) {
```

```
            revert CampaignInfoInvalidPlatformUpdate(platformHash, selection);
        }
        s_selectedPlatformHash[platformHash] = selection;
        emit CampaignInfoSelectedPlatformUpdated(platformHash, selection);
    }
```

Listing 3.3: `CampaignInfo::updateSelectedPlatform()`

Recommendation Timely update the platform fee when it is updated for a campaign.

Status This issue has been fixed in the following PR: `13`.

3.4 Improved Validation of Function Arguments

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple Contracts`
- 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 CC protocol is no exception. Specifically, if we examine the `CampaignInfo` contract, it has defined a number of campaign-wide risk parameters, such as `launchTime` and `deadline`. In the following, we show the corresponding routines that allow for their changes.

```
    function updateLaunchTime(
        uint256 launchTime
    )
        external
        override
        onlyOwner
        currentTimeIsLess(getLaunchTime())
        whenNotPaused
        whenNotCancelled
    {
        if (launchTime < block.timestamp && getDeadline() <= launchTime) {
            revert CampaignInfoInvalidInput();
        }
        s_campaignData.launchTime = launchTime;
        emit CampaignInfoLaunchTimeUpdated(launchTime);
    }

    /**
     * @inheritdoc ICampaignInfo
     */
    function updateDeadline(
        uint256 deadline
    )
        external
        override
        onlyOwner
        currentTimeIsLess(getLaunchTime())
        whenNotPaused
        whenNotCancelled
    {
        if (deadline <= getLaunchTime()) {
            revert CampaignInfoInvalidInput();
        }

        s_campaignData.deadline = deadline;
        emit CampaignInfoDeadlineUpdated(deadline);
    }
```

Listing 3.4: Example Setters in CampaignInfo

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis 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. Also, these parameters need to be properly enforced. For example, the above `updateLaunchTime()` should validate the given `launchTime` as follows: `if (launchTime < block.timestamp || getDeadline()<= launchTime)`, not current `if (launchTime < block.timestamp && getDeadline()<= launchTime)` (line 344).

Recommendation Properly enforce these system-wide parameters to ensure they are always maintained in an appropriate range. In addition to the above `updateLaunchTime()` routine, another `createCampaign()` routine can also be similarly improved. Moreover, a getter routine `GlobalParams::getPlatformDataOwner()` can be improved by removing the attached `platformIsListed(platformHash)` modifier (line 264).

Status This issue has been fixed in the following PR: 13.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the audited `CC` protocol, there exist a certain privileged account `owner` (or `admin`) that plays critical roles in governing and regulating the system-wide operations. It also has the privilege to regulate or govern the flow of assets within the protocol. In the following, we show representative privileged operations in the protocol.

```
    function updateProtocolAdminAddress(
        address protocolAdminAddress) external override onlyOwner notAddressZero(
            protocolAdminAddress) {
        s_protocolAdminAddress = protocolAdminAddress;
        emit ProtocolAdminAddressUpdated(protocolAdminAddress);}

    /**
     * @inheritdoc IGlobalParams
     */
    function updateTokenAddress(
        address tokenAddress) external override onlyOwner notAddressZero(tokenAddress) {
        s_tokenAddress = tokenAddress;
        emit TokenAddressUpdated(tokenAddress);}

    /**
     * @inheritdoc IGlobalParams
     */
    function updateProtocolFeePercent(
        uint256 protocolFeePercent) external override onlyOwner {
        s_protocolFeePercent = protocolFeePercent;
        emit ProtocolFeePercentUpdated(protocolFeePercent);}
```

Listing 3.5: Example Privileged Functions in `GlobalParams`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts 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 protocol users.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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

Status This issue has been mitigated as the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the CC protocol, which is a decentralized crowdfunding protocol designed to help creators launch and manage campaigns across multiple platforms. By providing a standardized infrastructure, the protocol simplifies the process of creating, funding, and managing crowdfunding initiatives in Web3 across different platforms. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.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 CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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