SMART CONTRACT AUDIT REPORT

for

Coresky DAO

Prepared By: Xiaomi Huang

PeckShield

October 22, 2024

Document Properties

Client	Coresky
Title	Smart Contract Audit Report
Target	Coresky
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	October 22, 2024	Xuxian Jiang	Final Release
1.0-rc	September 18, 2024	Xuxian Jiang	Release Candidate

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 Coresky DAO 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 Coresky

The Web3 industry has been experiencing significant growth. Coresky innovatively employs NFTs as investment certificates, focusing on the primary and semi-primary markets. By fragmenting early investment amounts of high-quality projects, Coresky enables community fundraising to share high-return investments with small and medium-sized investors. Additionally, Coresky plans to introduce a staking service for users to access working capital while retaining NFT equity. This audit covers its governance protocol and related contracts. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of Coresky DAO

Item	Description
Name	Coresky
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	October 22, 2024

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

- https://github.com/Coreskyofficial/csky-dao-contract.git (3ab612f)

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

- https://github.com/Coreskyofficial/csky-dao-contract.git (3ab612f)

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 classified into four categories accordingly, 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

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

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.

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 implementation of the staking support in `Coresky`. 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 logic, 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	3	■ ■ ■
Informational	0	
Total	4	

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 that 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, no ERC20 compliance issue was found and our detailed checklist can be found in Section ??. While there is no critical or high severity issue, the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability and 4 medium-severity vulnerabilities.

Table 2.1: Key Coresky Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Presale Refund Logic in AllocationUpgradeable	Business Logic	Resolved
PVE-002	High	Possible Denial-of-Service in Allocation Refund	Business Logic	Confirmed
PVE-003	Medium	Improved Validation of Function Arguments	Coding Practices	Confirmed
PVE-004	Medium	Incorrect apNftMint() Logic in CoreskyHubUpgradeable	Business Logic	Confirmed
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 Presale Refund Logic in AllocationUpgradeable

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AllocationUpgradeable`
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

Each token presale managed in `AllocationUpgradeable` supports the possibility of refunding user buys in case the presale is not successful. While examining the refunding logic, we notice current implementation should be improved.

In the following, we show the implementation of the related refund routine, i.e., `presaleRefund()`. When there is a long list of users, the refund process attempts to limit each refund call to maximum 1000 users. However, for each refund, it always validates the contract balance is no less than the total received sales (lines 468 and 483). Moreover, the storage state of `withdrawalAllocationTotalAmount` should be updated only with the sum of refunded users' `totalPayment`, not total received sales (line 496).

```
444     function presaleRefund(uint256 roundID, address payable _Referrer, uint256
            _ReferrerFee) public payable nonReentrant onlyRole(OPERATOR_ROLE){
445
446         require(roundID > 0, "project is empty");
447         // Verify time
448         if(fundraisingStatus[roundID] != Types.FundraisingStatus.Fail){
449             return;
450         }
451
452         // Get the project associated with the given roundID
453         Types.Project storage project = round[roundID];
454         Types.PreSaleLog[] memory _logs = preSaleLog[roundID];
455
```

```
        uint256 limit = 1000;
        uint256 lastLogs = _logs.length - refundIndex[roundID];
        if(limit > lastLogs){
            limit = lastLogs;
        }

        if (refundIndex[roundID] >= _logs.length) {
            return;
        }

        uint256 total = project.nftPrice * project.totalSales;
        if (project.payment == address(0)) {
            require(address(this).balance >= total, "Insufficient amount token");
            // Iterate over each recipient and transfer the corresponding amount of
                tokens
            uint256 i;
            for (; i < limit; i++) {
                Types.PreSaleLog memory _log = _logs[refundIndex[roundID]];
                // Record the total payment for the preSaleID of the sender
                uint256 totalPayment = project.preSaleRecords[_log.preSaleUser][_log.
                    preSaleID];
                _refundEth(roundID, _log.preSaleID, _Referrer,_ReferrerFee,_log.
                    preSaleUser,totalPayment);

                refundIndex[roundID]++;
            }
        } else {
            require(msg.value == 0, "Needn't pay mainnet token");
            // Iterate over each recipient and transfer the corresponding amount of
                tokens
            IERC20 _token = IERC20(project.payment);
            require(_token.balanceOf(address(this)) >= total, "Insufficient amount token
                ");
            uint256 i;
            for (; i < limit; i++) {
                 Types.PreSaleLog memory _log = _logs[refundIndex[roundID]];
                // Record the total payment for the preSaleID of the sender
                uint256 totalPayment = project.preSaleRecords[_log.preSaleUser][_log.
                    preSaleID];
                _refundTT(roundID, _log.preSaleID, _Referrer, _ReferrerFee, project.
                    payment, _log.preSaleUser, totalPayment);

                refundIndex[roundID]++;

            }
        }

        withdrawalAllocationTotalAmount[roundID] = total;

        emit PresaleRefund(roundID);
    }
```

Listing 3.1: `AllocationUpgradeable::presaleRefund()`

Recommendation Revise the above routine to properly refund presale users.

Status The issue has been fixed by this commit: `0275b6a`.

3.2 Possible Denial-of-Service in Allocation Refund

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `AllocationUpgradeable`
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

As mentioned in Section 3.1, each token presale allows for refunding user buys in case the presale is not successful. While examining the low-level logic to refund users, we notice the refund of native coins may lead to a denial-of-service issue that breaks the intended refund.

In the following, we shows the implementation of the native-coin-refunding routine, i.e., `_refundEth()`. While the refund amount has been properly deducted with the fee, the actual routine to send the fund back is performed in `TransferETH()`, which employs `assert(payable(_receiver).send(_Amount))` to ensure the refund is properly delivered over. However, if there is a receiver who attempts to decline the transfer, the entire refund process will be blocked. To fix, there is a need to use the low-level `call()` as documented in `https://consensys.io/diligence/blog/2019/09/stop-using-soliditys-transfer-now`.

```
    function _refundEth(uint256 roundID, uint256 preSaleID, address payable _Referrer,
        uint256 _ReferrerFee, address receiver, uint256 totalPayment) internal virtual{
        /* Amount that will be received by user (for Ether). */
        uint256 receiveAmount = totalPayment;
        // Referrer Fee
        uint256 referrerFee;
        if (_Referrer!=address(0) && _ReferrerFee > 0) {
            referrerFee = SafeMath.div(SafeMath.mul(_ReferrerFee, totalPayment),
                INVERSE_BASIS_POINT);
            receiveAmount = SafeMath.sub(receiveAmount, referrerFee);
            TransferETH(payable(_Referrer), referrerFee);
        }

        TransferETH(payable(receiver), receiveAmount);

        emit Refund(roundID, preSaleID, receiver, totalPayment, referrerFee,
            receiveAmount, block.timestamp);
    }

    function TransferETH(address payable _receiver, uint256 _Amount) internal {
        assert(payable(_receiver).send(_Amount));
```

```
    }
```

Listing 3.2: `AllocationUpgradeable::_refundEth()`

Recommendation Revise the above routine to properly refund presale users without being blocked.

Status The issue has been confirmed.

3.3 Improved Validation of Function Arguments

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Rentable`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Coresky` governance protocol is no exception. Specifically, if we examine the `AllocationUpgradeable` contract, it has defined a number of protocol-wide risk parameters, such as `voteEndTime` and `mintEndTime`. In the following, we show the corresponding routines that allow for their changes.

```
    function setEndTime(uint256 _roundID, uint256 _endTime) public onlyRole(
        OPERATOR_ROLE) {
        Types.Project storage project = round[_roundID];
        require(project.target != address(0), "Project does not exist");
        project.endTime = _endTime;
    }

    // Set Project - Second Voting End Time
    function setVoteEndTime(uint256 _roundID, uint256 _voteEndTime) public onlyRole(
        OPERATOR_ROLE) {
        Types.Project storage project = round[_roundID];
        require(project.target != address(0), "Project does not exist");
        voteEndTime[_roundID] = _voteEndTime;
    }

    // Set project - mint end time
    function setMintEndTime(uint256 _roundID, uint256 _mintEndTime) public onlyRole(
        OPERATOR_ROLE) {
        Types.Project storage project = round[_roundID];
        require(project.target != address(0), "Project does not exist");
        mintEndTime[_roundID] = _mintEndTime;
```

```
    }
```

Listing 3.3: Example Setters in AllocationUpgradeable

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, both `sendFundraising()` and `withdrawalAllocation()` routines can be improved by enforcing the following requirements: `require(voteEndTime[roundID] <= block.timestamp)` and `require(project.endTime <= block.timestamp)`.

Recommendation Properly enforce these system-wide parameters to ensure they are always maintained in an appropriate range.[1]

Status The issue has been confirmed.

3.4 Incorrect apNftMint() Logic in CoreskyHubUpgradeable

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CoreskyHubUpgradeable`
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

The `Coresky` governance protocol has a core `CoreskyHubUpgradeable` contract to oversee the presale process. In the process of examining the presale-associated `NFT` logic, we notice current implementation should be improved.

In the following, we show the implementation of the related `NFT`-minting routine, i.e., `apNftMint()`. It has a rather straightforward logic in calculating and minting the total number of `NFT`s to mint, i.e., `lastMintNum`. However, the `lastMintNum` calculation should be revisited when the total `NFT`s to mint is not larger than the minted number. In other words, when all `NFT`s were already minted, the calculated `lastMintNum` should be 0, not the full `NFT` amount one more time (line 530).

```
523     function apNftMint(Types.EIP712Signature calldata signature, uint256 roundID) public
            nonReentrant {...
524         uint256 lastMintNum;
525         uint256 mintNum = IAllocation(targetAllocation).getMintNum(user, roundID);
```

[1] The `withdrawalAllocation()` routine can also be improved by properly updating `withdrawalAllocationTo[_roundID][_to] += _amount` (line 984).

```
        if(totalPreSaleNum > mintNum){
            lastMintNum = totalPreSaleNum.sub(mintNum);
        } else {
            lastMintNum = totalPreSaleNum;
        }
        if(lastMintNum > maxMintLimit) {
            lastMintNum = maxMintLimit;
        }

        // valide sign
        MetaTxLibUpgradeable.validateApNftMintSignature(signature, roundID,
            targetAllocation, lastMintNum);
        // function batchMint(address _to, uint256 _amount) external;
        IApNFT(apnft).batchMint(user, lastMintNum);
        // set NFT-mint num
        mintNum = mintNum.add(lastMintNum);
        IAllocation(targetAllocation).setMintNum(roundID, user, mintNum);
        emit Events.ApNFTMint(roundID, apnft, user, totalPreSaleNum, mintNum, block.
            timestamp);

    }
```

Listing 3.4: `CoreskyHubUpgradeable::apNftMint()`

Recommendation Revise the above routine to properly calculate the correct `NFTs` to mint.

Status The issue has been confirmed.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Coresky` DAO contracts, there is a privileged account `owner` (or with `OPERATOR_ROLE` role) that plays a critical role in governing and regulating the governance-wide operations (e.g., configure parameters and assign roles). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contract.

```
    function setFeeTo(address _feeTo) external onlyRole(OPERATOR_ROLE) {
        StorageLib.setFeeTo(_feeTo);
```

```
    }
    /**
     * @dev Set the allocation fundraising fee.
     *
     * @param _fee The new allocation fundraising fee to set.
     */
    function setFee(uint256 _fee) external onlyRole(OPERATOR_ROLE) {
        StorageLib.setFee(_fee);
    }
    /**
     * @dev Set the allocation refund fee address.
     *
     * @param _feeTo The new allocation refund fee address to set.
     */
    function setBackFeeTo(address _feeTo) external onlyRole(OPERATOR_ROLE) {
        StorageLib.setBackFeeTo(_feeTo);
    }
    /**
     * @dev Set the allocation refund fee.
     *
     * @param _fee The new allocation refund fee to set.
     */
    function setBackFee(uint256 _fee) external onlyRole(OPERATOR_ROLE) {
        StorageLib.setBackFee(_fee);
    }

     /**
     * @dev Set the allocation pause.
     *
     * @param _roundID The ID of the presale round.
     */
    function pause(uint256 _roundID) public onlyRole(OPERATOR_ROLE)  {
        address targetAllocation = _allocation(_roundID);
        __checkAlloction(targetAllocation);
        IAllocation(targetAllocation).pause(_roundID);
    }

     /**
     * @dev Set the allocation unpause.
     *
     * @param _roundID The ID of the presale round.
     */
    function unpause(uint256 _roundID) public onlyRole(OPERATOR_ROLE)  {
        address targetAllocation = _allocation(_roundID);
        __checkAlloction(targetAllocation);
        IAllocation(targetAllocation).unpause(_roundID);
    }
```

Listing 3.5: Example Privileged Operations in `CoreskyHubUpgradeable`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it would be worrisome if the privileged account is a plain `EOA` account. Note that a multi-sig account could greatly alleviate this concern, though 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

In the meantime, various DAO contracts make use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

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 mitigated as the team plans to use a multi-sig to have the `owner` role.

4 Conclusion

In this security audit, we have examined the Coresky contract design and related DAO implementation. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities and found no issue in these areas. We then proceeded to examine other areas such as coding practices and business logics. Overall, no issue was found in these areas, and the current deployment follows the best practice. Meanwhile, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

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 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] 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.