SMART CONTRACT AUDIT REPORT

for

Coresky Token & DAO

Prepared By: Xiaomi Huang

PeckShield

September 18, 2024

Document Properties

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

Version Info

Version	Date	Author	Description
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 4
- 1.1 About Coresky . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 6

2 Findings 8
- 2.1 Summary . . . 8
- 2.2 Key Findings . . . 9

3 ERC20 Compliance Checks 10

4 Detailed Results 13
- 4.1 Incorrect Presale Refund Logic in AllocationUpgradeable . . . 13
- 4.2 Possible Denial-of-Service in Allocation Refund . . . 15
- 4.3 Improved Validation of Function Arguments . . . 16
- 4.4 Incorrect apNftMint() Logic in CoreskyHubUpgradeable . . . 17
- 4.5 Trust Issue of Admin Keys . . . 18

5 Conclusion 21

References 22

1 Introduction

Given the opportunity to review the design document and related source code of the Coresky token and related governance contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract exhibits no ERC20 compliance issues or security concerns. 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 protocol token Csky and as well as its governance contracts. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of Coresky Token Contract

Item	Description
Name	Coresky
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	September 18, 2024

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

- https://github.com/Coreskyofficial/csky-token-contract.git (cb2ce18)

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

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

- https://github.com/Coreskyofficial/csky-token-contract.git (TBD)
- https://github.com/Coreskyofficial/csky-dao-contract.git (TBD)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

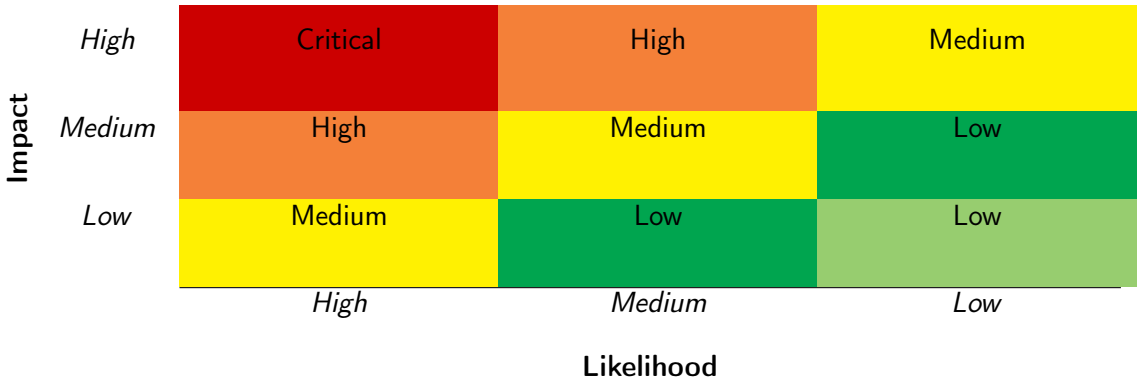
- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.2: Vulnerability Severity Classification



To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead of Transfer
	Costly Loop
	(Unsafe) Use of Untrusted Libraries
	(Unsafe) Use of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Coresky token contract and its related DAO contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	4	■■■■
Low	0	
Total	5	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found and our detailed checklist can be found in Section 3. 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	
PVE-002	High	Possible Denial-of-Service in Allocation Refund	Business Logic	
PVE-003	Medium	Improved Validation of Function Arguments	Coding Practices	
PVE-004	Medium	Incorrect apNftMint() Logic in CoreskyHubUpgradeable	Business Logic	
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	

Besides recommending specific countermeasures to mitigate the above issue(s), we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

3 | ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20 -compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited Coresky token contract. In the surrounding two tables, we outline the respective list of basic view-only functions (Table 3.1) and key state-changing functions (Table 3.2) according to the widely-adopted ERC20 specification.

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements, but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional Opt-in Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Upgradable	The token contract allows for future upgrades	—
Whitelistable	The token contract allows the owner or privileged users to whitelist a specific address such that only token transfers and other operations related to that address are allowed	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	—

4 Detailed Results

4.1 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
```

```
456         uint256 limit = 1000;
457         uint256 lastLogs = _logs.length - refundIndex[roundID];
458         if(limit > lastLogs){
459             limit = lastLogs;
460         }
461 
462         if (refundIndex[roundID] >= _logs.length) {
463             return;
464         }
465 
466         uint256 total = project.nftPrice * project.totalSales;
467         if (project.payment == address(0)) {
468             require(address(this).balance >= total, "Insufficient amount token");
469             // Iterate over each recipient and transfer the corresponding amount of
                    tokens
470             uint256 i;
471             for (; i < limit; i++) {
472                 Types.PreSaleLog memory _log = _logs[refundIndex[roundID]];
473                 // Record the total payment for the preSaleID of the sender
474                 uint256 totalPayment = project.preSaleRecords[_log.preSaleUser][_log.
                        preSaleID];
475                 _refundEth(roundID, _log.preSaleID, _Referrer,_ReferrerFee,_log.
                        preSaleUser,totalPayment);
476 
477                 refundIndex[roundID]++;
478             }
479         } else {
480             require(msg.value == 0, "Needn't pay mainnet token");
481             // Iterate over each recipient and transfer the corresponding amount of
                    tokens
482             IERC20 _token = IERC20(project.payment);
483             require(_token.balanceOf(address(this)) >= total, "Insufficient amount token
                    ");
484             uint256 i;
485             for (; i < limit; i++) {
486                  Types.PreSaleLog memory _log = _logs[refundIndex[roundID]];
487                 // Record the total payment for the preSaleID of the sender
488                 uint256 totalPayment = project.preSaleRecords[_log.preSaleUser][_log.
                        preSaleID];
489                 _refundTT(roundID, _log.preSaleID, _Referrer, _ReferrerFee, project.
                        payment, _log.preSaleUser, totalPayment);
490 
491                 refundIndex[roundID]++;
492 
493             }
494         }
495 
496         withdrawalAllocationTotalAmount[roundID] = total;
497 
498         emit PresaleRefund(roundID);
499     }
```

Listing 4.1: `AllocationUpgradeable::presaleRefund()`

Recommendation Revise the above routine to properly refund presale users.

Status

4.2 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 4.1, 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`.

```
501     function _refundEth(uint256 roundID, uint256 preSaleID, address payable _Referrer,
            uint256 _ReferrerFee, address receiver, uint256 totalPayment) internal virtual{
502         /* Amount that will be received by user (for Ether). */
503         uint256 receiveAmount = totalPayment;
504         // Referrer Fee
505         uint256 referrerFee;
506         if (_Referrer!=address(0) && _ReferrerFee > 0) {
507             referrerFee = SafeMath.div(SafeMath.mul(_ReferrerFee, totalPayment),
                    INVERSE_BASIS_POINT);
508             receiveAmount = SafeMath.sub(receiveAmount, referrerFee);
509             TransferETH(payable(_Referrer), referrerFee);
510         }
511
512         TransferETH(payable(receiver), receiveAmount);
513
514         emit Refund(roundID, preSaleID, receiver, totalPayment, referrerFee,
                receiveAmount, block.timestamp);
515     }
516
517     function TransferETH(address payable _receiver, uint256 _Amount) internal {
518         assert(payable(_receiver).send(_Amount));
```

```
519     }
```

Listing 4.2: AllocationUpgradeable::_refundEth()

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

Status

4.3 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.

```
867     function setEndTime(uint256 _roundID, uint256 _endTime) public onlyRole(
            OPERATOR_ROLE) {
868         Types.Project storage project = round[_roundID];
869         require(project.target != address(0), "Project does not exist");
870         project.endTime = _endTime;
871     }
872
873     // Set Project - Second Voting End Time
874     function setVoteEndTime(uint256 _roundID, uint256 _voteEndTime) public onlyRole(
            OPERATOR_ROLE) {
875         Types.Project storage project = round[_roundID];
876         require(project.target != address(0), "Project does not exist");
877         voteEndTime[_roundID] = _voteEndTime;
878     }
879
880     // Set project - mint end time
881     function setMintEndTime(uint256 _roundID, uint256 _mintEndTime) public onlyRole(
            OPERATOR_ROLE) {
882         Types.Project storage project = round[_roundID];
883         require(project.target != address(0), "Project does not exist");
884         mintEndTime[_roundID] = _mintEndTime;
```

```
885     }
```

Listing 4.3: 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

4.4 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).

```
526
527         if(totalPreSaleNum > mintNum){
528             lastMintNum = totalPreSaleNum.sub(mintNum);
529         } else {
530             lastMintNum = totalPreSaleNum;
531         }
532         if(lastMintNum > maxMintLimit) {
533             lastMintNum = maxMintLimit;
534         }
535
536         // valide sign
537         MetaTxLibUpgradeable.validateApNftMintSignature(signature, roundID,
                targetAllocation, lastMintNum);
538         // function batchMint(address _to, uint256 _amount) external;
539         IApNFT(apnft).batchMint(user, lastMintNum);
540         // set NFT-mint num
541         mintNum = mintNum.add(lastMintNum);
542         IAllocation(targetAllocation).setMintNum(roundID, user, mintNum);
543         emit Events.ApNFTMint(roundID, apnft, user, totalPreSaleNum, mintNum, block.
                timestamp);
544
545     }
```

Listing 4.4: `CoreskyHubUpgradeable::apNftMint()`

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

Status

4.5 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` token and related 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, assign roles, as well as pause/unpause/upgrade the token contract). 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.

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

```
307     }
308     /**
309      * @dev Set the allocation fundraising fee.
310      *
311      * @param _fee The new allocation fundraising fee to set.
312      */
313     function setFee(uint256 _fee) external onlyRole(OPERATOR_ROLE) {
314         StorageLib.setFee(_fee);
315     }
316     /**
317      * @dev Set the allocation refund fee address.
318      *
319      * @param _feeTo The new allocation refund fee address to set.
320      */
321     function setBackFeeTo(address _feeTo) external onlyRole(OPERATOR_ROLE) {
322         StorageLib.setBackFeeTo(_feeTo);
323     }
324     /**
325      * @dev Set the allocation refund fee.
326      *
327      * @param _fee The new allocation refund fee to set.
328      */
329     function setBackFee(uint256 _fee) external onlyRole(OPERATOR_ROLE) {
330         StorageLib.setBackFee(_fee);
331     }
332
333      /**
334      * @dev Set the allocation pause.
335      *
336      * @param _roundID The ID of the presale round.
337      */
338     function pause(uint256 _roundID) public onlyRole(OPERATOR_ROLE)  {
339         address targetAllocation = _allocation(_roundID);
340         __checkAlloction(targetAllocation);
341         IAllocation(targetAllocation).pause(_roundID);
342     }
343
344      /**
345      * @dev Set the allocation unpause.
346      *
347      * @param _roundID The ID of the presale round.
348      */
349     function unpause(uint256 _roundID) public onlyRole(OPERATOR_ROLE)  {
350         address targetAllocation = _allocation(_roundID);
351         __checkAlloction(targetAllocation);
352         IAllocation(targetAllocation).unpause(_roundID);
353     }
```

Listing 4.5: 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

5 | 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] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

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