Security Audit

Celo (DeFi)

October 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

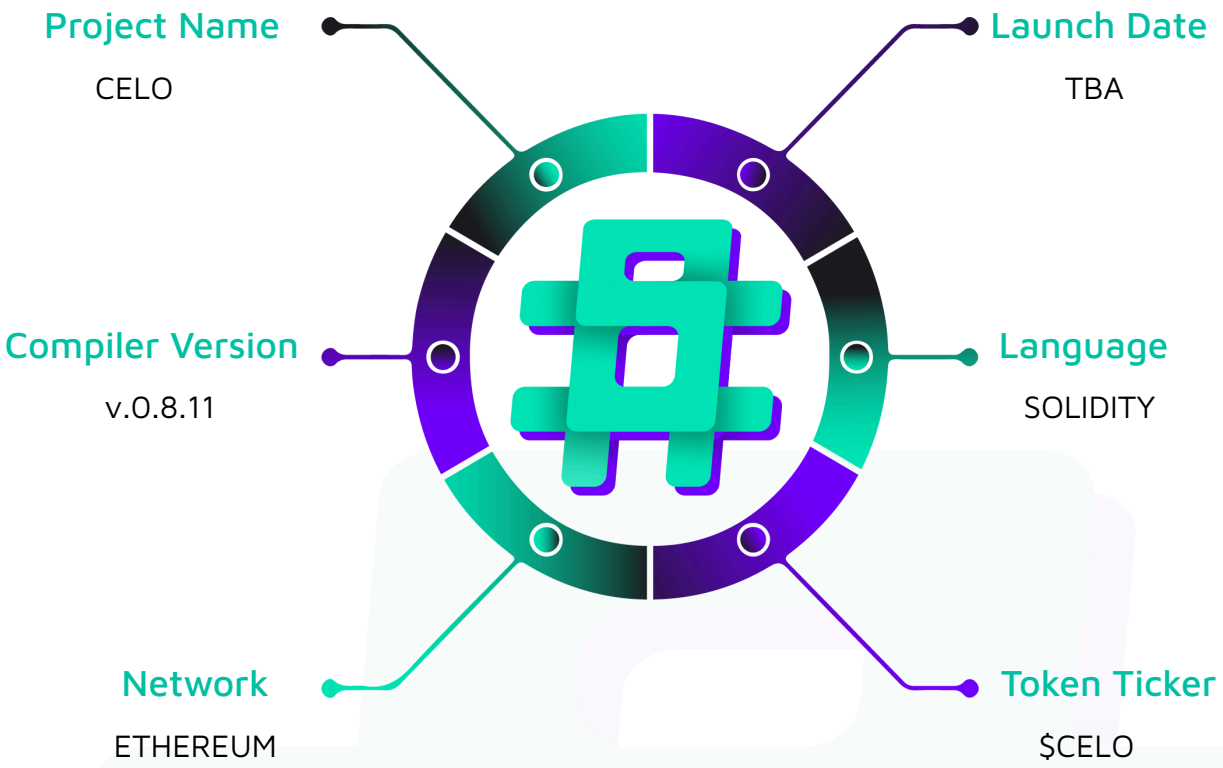
The Celo team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

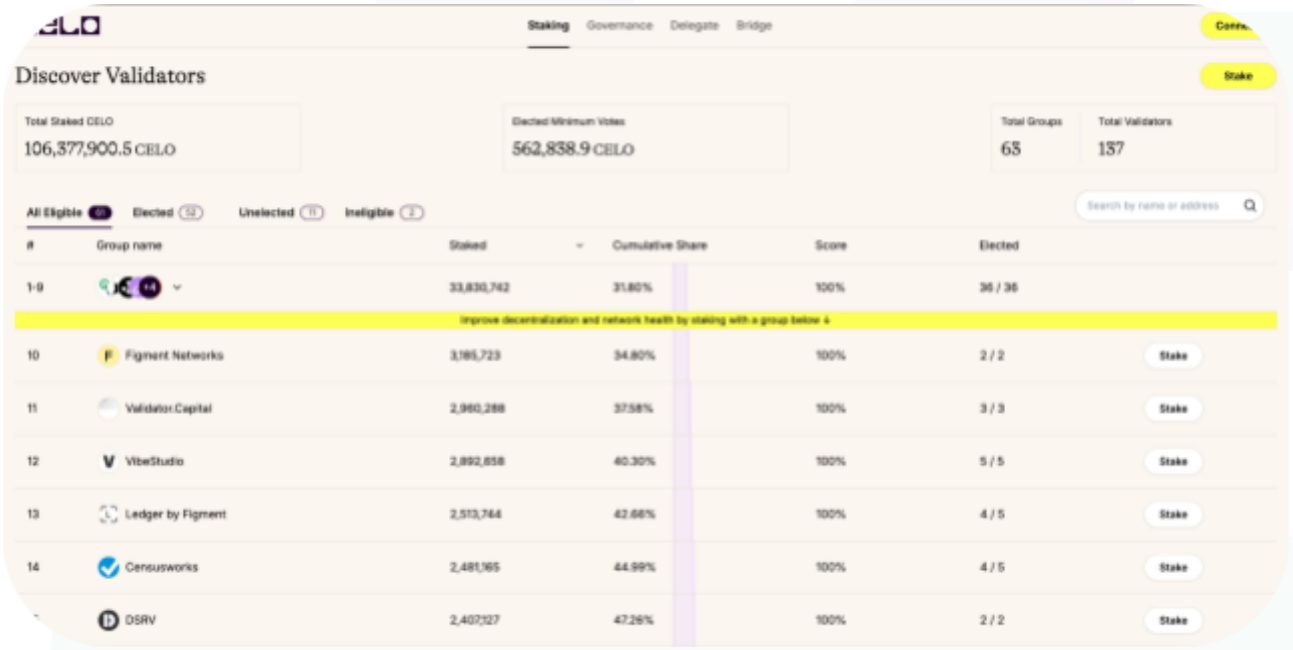
The Celo project is a purpose-driven, Ethereuqm-anchored Layer-2 blockchain platform designed to enable fast, low-cost, and carbon-aware payments and decentralized finance (DeFi) applications globally. It supports gas payments in ERC-20 tokens and incorporates modular technologies, including an OP-Stack L2, EigenDA for scalable data availability, and a zkEVM (via Succinct SP1) for verified execution, together enabling one-second block times and sub-cent fees. By prioritizing accessibility, interoperability with Ethereum, and sustainability (for example, allocating 20 % of transaction fees to carbon offsets), Celo positions itself as an infrastructure foundation for real-world financial inclusion and on-chain economic growth.

Project Name: Celo
Project Type: DeFi
Compiler Version: 0.8.11
Website: https://celo.org/
Logo:

Visualised Context:



Project Visuals:



Audit Scope

We at Hashlock audited the solidity code within the Celo project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Celo Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**October, 2025**
Contract 1	Account.sol
Contract 2	DefaultStrategy.sol
Contract 3	GroupHealth.sol
Contract 4	Managed.sol
Contract 5	Manager.sol
Contract 6	Pausable.sol
Contract 7	SpecificGroupStrategy.sol
Audited GitHub Commit Hash	a3c350a23ed3aade246a2bb0a93ec53a7607a5be
Fix Review GitHub Commit Hash	9c85724899f1f0185d9d57bbd8e602e1b6a7f710

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

1 Medium-severity vulnerability

3 Low-severity vulnerabilities

1 Gas Optimisation

4 QAs

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
Account.sol A contract that facilitates CELO voting and withdrawals, acting as an account proxy managed by a separate Manager contract. Allows the owner (and designated Manager) to: - Initialize the contract, linking it to the Celo Registry and Manager - Set the pauser address and pause/unpause contract functions - Schedule CELO to be voted for by validator groups - Schedule transfers of votes between validator groups - Schedule CELO withdrawals for specific beneficiaries - Vote on governance proposals - Configure voting parameters Allows the user to: - Activate pending votes and vote any scheduled/unlocked CELO - Execute scheduled vote revocations from a group - Start a scheduled withdrawal for a beneficiary, which processes immediate withdrawals and starts the unlock period for locked funds - Finalize and claim a pending withdrawal after the unlock period has passed	**Contract achieves this functionality.**

DefaultStrategy.sol Manages CELO vote distribution for the default staking pool, balancing votes across a sorted list of active, healthy validator groups. Allows the owner to: - Initialize the contract, set dependencies (Account, GroupHealth, etc.), and set the pauser - Configure parameters like distribution limits and sorting loop limits - Add groups to the "activatable" list - Forcibly deactivate groups and set the minimum active group count - Manually update stCELO accounting for a group Allows the Manager or SpecificGroupStrategy contract to: - Get a calculated vote distribution for new deposits - Get a calculated withdrawal distribution Allows the user to: - Activate a group from the activatable list if it's healthy - Deactivate a group if the GroupHealth contract reports it as unhealthy - Trigger a rebalance of stCELO between an over-funded and under-funded group - Help re-sort the active group list if it becomes unsorted	**Contract achieves this functionality.**
GroupHealth.sol A helper contract that stores and updates the health status of Celo validator groups, checking their	**Contract achieves this functionality.**

registration, slashing status, and whether they have at least one elected member. Allows the owner to: - Initialize the contract, linking it to the Celo Registry - Set the pauser address (and subsequently pause/unpause the contract) Allows the user to: - Trigger a comprehensive health check for a group, which internally verifies its validity and searches for an elected member (updateGroupHealth) - Mark a group as healthy by providing specific indexes as "hints" to prove a member is currently elected, which can be more gas-efficient (markGroupHealthy)	
Managed.sol An abstract, upgradeable base contract used via inheritance to provide a simple Manager role for access control, separate from the Owner. Allows the owner to: - Initialize the contract by setting the initial manager address - Set or change the manager's address Provides for inheriting contracts: - An onlyManager modifier to restrict specific functions, making them callable only by the designated manager address	**Contract achieves this functionality.**

Manager.sol A central coordinator contract for the StakedCelo system that handles deposits/withdrawals, routes CELO to different voting strategies, and manages the minting/burning of stCELO tokens. Allows the owner to: - Initialize the contract and set all critical contract dependencies (like StakedCelo, Account, GroupHealth, and strategies) - Set the pauser address - Forcibly change the voting strategy for any user. Allows the StakedCelo contract to: - Call the transfer hook during a token transfer, which rebalances votes if the sender and receiver have different strategies Allows the Strategy contracts to: - Schedule vote transfers between validator groups as part of their internal logic Allows the user to: - Deposit CELO (which mints stCELO) and have the CELO automatically voted according to their chosen strategy - Withdraw (burn) stCELO to schedule a CELO withdrawal from the system - Change their own voting strategy (e.g., switch between the default pool and a specific validator) - Vote on governance proposals, which lock their stCELO	**Contract achieves this functionality.**

- Revoke votes on governance proposals and unlock their stCELO - Trigger public rebalancing functions to help correct vote distribution among groups	
Pausable.sol An abstract contract that provides emergency stop (pause and unpause) functionality to be inherited by other contracts, all managed by a special "Pauser" role. Allows the Pauser to: - Pause the contract - Unpause the contract Allows any user to: - Check if the contract is currently paused - View the address of the Pauser Provides for inheriting contracts: - An onlyWhenNotPaused modifier to block functions during a pause - An internal _setPauser function for the inheriting contract to assign the Pauser role (typically done by its Owner)	**Contract achieves this functionality.**
SpecificGroupStrategy.sol A contract that manages CELO vote distribution for users who select a single, specific validator group. It automatically handles "overflow" (when a group is full) and "unhealthy" states by redirecting those funds to the DefaultStrategy pool.	**Contract achieves this functionality.**

Allows the owner to: - Initialize the contract, set dependencies (Account, GroupHealth, DefaultStrategy), and set the pauser - Block specific validator groups from being chosen by users - Unblock previously blocked groups - Manually adjust a group's stCELO accounting Allows the Manager contract to: - Get a deposit distribution, which routes CELO to the user's chosen group or (if unhealthy/full) overflows it to the DefaultStrategy - Get a withdrawal distribution, which pulls CELO from both the specific group and any associated funds in the DefaultStrategy Allows the user to: - Call rebalanceWhenHealthChanged to move funds between the default pool and their specific group as its health status changes - Call rebalanceOverflowedGroup to move funds from the default pool back to their specific group if it gains new voting capacity	

Code Quality

This audit scope involves the smart contracts of the Celo, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Celo smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] Contracts - Missing Storage Gap in UUPSUpgradeable Contracts May Cause Storage Collisions

Description

The `UUPSUpgradeable` contracts in `Account.sol`, `DefaultStrategy.sol`, `GroupHealth.sol`, `Manager.sol` and `SpecificGroupStrategy.sol` lack a reserved storage gap (`__gap`), which is essential for maintaining upgrade safety. Without it, adding new variables in future implementations can cause storage collisions, potentially overwriting existing data and leading to unpredictable behavior.

Vulnerability Details

The UUPS (Universal Upgradeable Proxy Standard) pattern enables contract upgrades while retaining the existing state. However, it demands strict control over the storage layout. During an upgrade:

- The storage layout of the new implementation must be compatible with the existing storage.
- Adding new variables to a parent contract will push down all child contract variables in storage.

In the current implementation, for instance, `Account.sol`:

```
contract Account is UUPSOwnableUpgradeable, UsingRegistryUpgradeable, Managed, IAccount, Pausable {

   //@audit: No __gap variable is defined
```

`Account.sol` does not implement its own storage gap. If future versions of the contract add new storage variables, storage `collisions` might happen.

Impact

Without a reserved storage gap, newly introduced state variables in upgraded implementations may conflict with storage slots already used by inherited contracts. This overlap can corrupt the contract's state, disrupt its logic, or create subtle, hard-to-detect vulnerabilities.

Recommendation

Add a fixed-size storage gap at the end of the contract, as recommended by OpenZeppelin's UUPSUpgradeable pattern. For example:

```
uint256[50] private __gap;
```

Status

Resolved

Low

[L-01] Contracts - Important functions lack event emissions

Description

In Solidity, events provide a critical logging mechanism that external observers (like dApps, users, or auditors) rely on to track what's happening on-chain. Important functions, such as those that modify contract state, transfer funds, or change ownership, should emit events to provide transparency.

The following critical functions were lacking in event emissions:

```
Account.sol::setAllowedToVoteOverMaxNumberOfGroups()

Account.sol::votePartially()

DefaultStrategy.sol::setDependencies()

DefaultStrategy.sol::setSortingParams()

DefaultStrategy.sol::generateDepositVoteDistribution()

DefaultStrategy.sol::rebalance()

DefaultStrategy.sol::generateWithdrawalVoteDistribution()

DefaultStrategy.sol::setMinCountOfActiveGroups()

DefaultStrategy.sol::addActivatableGroup()

DefaultStrategy.sol::updateGroupStCelo()

Manager.sol::withdraw()

Manager.sol::revokeVotes()

Manager.sol::updateHistoryAndReturnLockedStCeloInVoting()

Manager.sol::deposit()

Manager.sol::transfer()

Manager.sol::unlockBalance()

Manager.sol::rebalance()
```

```
Manager.sol::rebalanceOverflow()

Manager.sol::scheduleTransferWithinStrategy()

Manager.sol::voteProposal()

SpecificGroupStrategy.sol::updateGroupStCelo()

SpecificGroupStrategy.sol::generateWithdrawalVoteDistribution()

SpecificGroupStrategy.sol::generateDepositVoteDistribution()

SpecificGroupStrategy.sol::rebalanceWhenHealthChanged()

SpecificGroupStrategy.sol::rebalanceOverflowedGroup()
```

Impact

Critical actions (e.g., deposits, withdrawals) occur silently, making monitoring via block explorers or off-chain tools impossible.

Recommendation

Emit appropriate events in all important state-changing functions.

Status

Resolved

[L-02] Contracts- Unrestricted renounceOwnership() call

Description

The contracts `Account.sol`, `DefaultStrategy.sol`, `GroupHealth.sol`, `Managed.sol`, `Manager.sol` and `SpecificGroupStrategy.sol` inherit from `OwnableUpgradeable` contract, which includes the `renounceOwnership()` function. This function allows the current owner to irreversibly relinquish ownership, setting the owner address to `address(0)`.

Impact

In the current implementation, `renounceOwnership()` is not overridden or restricted, meaning any accidental or intentional call by the owner will permanently remove owner control over the contract.

Recommendation

If ownership should never be renounced (common for fixed-governance or permanently-administered contracts), override and disable it by adding the following function:

```
function renounceOwnership() public override onlyOwner {

    revert("Renouncing ownership is disabled.");

}
```

Status

Resolved

[L-03] GroupHealth.sol - Off-by-One Error in isGroupMemberElected

Description

In `GroupHealth.isGroupMemberElected()`, the check `if (index > currentNumberOfElectedValidators)` should be `if (index >= currentNumberOfElectedValidators)`. Array and list indices are 0-based, so an index equal to the length is out of bounds.

Impact

If index is equal to `currentNumberOfElectedValidators`, the check will pass, and the subsequent call to `validatorSignerAddressFromCurrentSet(index)` will likely revert due to an out-of-bounds access. This would cause the transaction to fail. The impact is low because this is an unlikely edge case, and it `would` fail safely.

Recommendation

Change the check to `if (index >= currentNumberOfElectedValidators)`.

Status

Resolved

QA

[Q-01] Contract - Non-Standard Naming Conventions

Description

Several private and internal functions and state variables deviate from the project's documented naming conventions, leading to inconsistent styling. This inconsistency increases the cognitive load when auditing or extending the contracts. It also raises the risk of introducing future mistakes due to misidentified function intents.

```
validatePendingWithdrawalRequest(...)
getAndUpdateToVoteAndToRevoke(...)
updateUnhealthyGroupStCelo(...)
transferToDefaultStrategy(...)
distributeVotes(...)
trySort(...)
```

Recommendation

Review all private and internal members and update their identifiers to follow the agreed naming standard (e.g., prefixing internal functions with _ and using lowerCamelCase for variables). Document the convention in the contributing guide to prevent regressions. Run linting or static analysis checks in CI to enforce compliance automatically.

Status

Acknowledged

[Q-02] GroupHealth.sol - Use of Magic Number

Description

In `GroupHealth._isGroupPartiallyValid()`, the number `10**24` is used to check the `slashMultiplier`. Magic numbers make the code harder to read and understand.

Recommendation

Define a constant with a descriptive name for this value, e.g., `uint256 private constant MIN_SLASH_MULTIPLIER = 10**24;`

Status

Acknowledged

[Q-03] GroupHealth.sol - Unused Variable Declarations

Description

Several legacy variables remain declared but unreferenced, e.g., contracts/Manager.sol (`activeGroups`) and contracts/Manager.sol (`deprecatedGroups`). contracts/DefaultStrategy.sol also keeps a `GroupWithVotes` struct that is never instantiated. These dead declarations complicate storage layout reviews and mislead contributors about still-active features.

Recommendation

Delete the unused declarations or reintegrate the logic that consumes them so the deployed storage layout reflects only the live state. Update documentation/comments accordingly to avoid suggesting unsupported flows. Representative leftovers:

Status

Acknowledged

[Q-04] Manager#_transferWithoutChecks - Vote Migration is Not Performed at This Layer (By Design)

Description

`_transferWithoutChecks` updates internal strategy accounting when a user switches strategies or transfers stCELO to an address using a different strategy. Although this routine does not immediately reschedule votes within the Account contract, this behavior is intentional.

To save gas and reduce on-chain overhead, the protocol does not migrate votes during each individual strategy change. Instead, vote movement is handled periodically by an off-chain automation component (the staked-CELO bot), which invokes rebalance to reconcile vote allocations across strategies. Because of this system design, _transferWithoutChecks omits direct calls such as scheduleTransfer or scheduleWithdrawals, and postpones vote migration until the next automated rebalance cycle.

Recommendation

No functional changes are required. However, adding more documentation in Manager.sol and including explicit assertions in unit tests—as the team already plans—will help make the intended vote-migration model clearer to future maintainers and auditors.

Status

Acknowledged

Gas

[G-01] GroupHealth#areGroupMembersElected - Unnecessary storage writes increase gas cost

Description

The function `areGroupMembersElected` temporarily writes each member's address into membersMappingHelper (a storage `mapping`) and later deletes them. Each SSTORE and subsequent DELETE incurs significant gas (first write ≈ 20k, delete gives a refund, but still net cost). This pattern is unnecessary for a transient membership check and increases per-call gas costs, especially when called frequently or when members or the validator set size is large.

```
function areGroupMembersElected(address[] memory members) private returns (bool) {
    for (uint256 j = 0; j < members.length; j++) {
        membersMappingHelper[members[j]] = true;
    }


    bool result;
    address validator;
    uint256 n = numberValidatorsInCurrentSet();
    for (uint256 i = 0; i < n; i++) {
        validator = validatorSignerAddressFromCurrentSet(i);
        if (membersMappingHelper[validator] == true) {
            result = true;
            break;
        }
    }

    for (uint256 j = 0; j < members.length; j++) {
        delete membersMappingHelper[members[j]];
    }
    return result;
}
```

Recommendation

Replace the temporary storage writes with an in-memory comparison (nested loops) to avoid `SSTORE`/`DELETE` operations. If `members.length` is small relative to the validator set, a nested loop (for each validator, scan the members array) will be cheaper and avoid persistent storage churn. For very large lists, consider off-chain proofs or a sorted structure to enable binary search. Example replacement (minimal drop-in body):

```
function areGroupMembersElected(address[] memory members) private returns (bool) {
    bool result;
    address validator;
    uint256 n = numberValidatorsInCurrentSet();

    // For each elected validator, scan the members array in memory (no storage writes).
    for (uint256 i = 0; i < n; i++) {
        validator = validatorSignerAddressFromCurrentSet(i);
        for (uint256 j = 0; j < members.length; j++) {
            if (validator == members[j]) {
                result = true;
                break;
            }
        }
        if (result) {
            break;
        }
    }

    return result;
}
```

Status

Acknowledged

Centralisation

Celo values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, Celo seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au