PUBLIC

Code Assessment

of the VoteDelegate Smart Contracts

April 24, 2025

Produced for

Sky

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Sky with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of VoteDelegate according to Scope to support you in forming an opinion on their security risks.

VoteDelegate implements a vote delegation system allowing governance token holders to delegate their voting power to delegates.

The most critical subjects covered in our audit are functional correctness, asset solvency and integration into the system. Security regarding all the aforementioned subjects is high.

The general subjects covered are usability and access control. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the VoteDelegate repository based on the documentation files.

2.1.1 *Maker VoteDelegate*

The scope consists of the two solidity smart contracts:

```
src/
    VoteDelegate.sol
    VoteDelegateFactory.sol
```

For the solidity smart contracts, the compiler version `0.8.16` was chosen and `evm_version` is set to `london`. Since (Version 2), compiler version `0.8.21` was chosen and `evm_version` is set to `shanghai`.

2.1.2 *Sky VoteDelegate*

The scope consists of the two solidity smart contracts:

```
src/
    VoteDelegate.sol
    VoteDelegateFactory.sol
```

For the solidity smart contracts, the compiler version `0.8.21` was chosen and `evm_version` is set to `shanghai`.

The table below indicates the code versions relevant to this report and when they were received.

Maker VoteDelegate

V	Date	Commit Hash	Note
1	04 Mar 2024	edf24ce9ba78aea0609fbaa0a7d1f6c24f1460fb	Initial Version
2	26 Apr 2024	9519c823b3e5ceed9a841f866eb034bcb4cbaf41	Second Version
3	03 July 2024	4484057a3c2d575edfc779cf432514ba83fccbe8	Finalization
4	15 Aug 2024	01e7b578366d6d88936c596996fba6d69d2d0753	Usage of Create

Sky VoteDelegate

V	Date	Commit Hash	Note
1	05 Mar 2025	ca1f2fb9b85ca2f3d89dc200766139c1613809ca	Initial Version
2	22 Apr 2025	ab978364103bbfcb17c40bc98649808b2d5cfea3	Gas Optimization

2.1.3 Excluded from scope

Any other files not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

2.2 System Overview

Vote delegation allows for governance token holders to delegate their voting power to delegates. These delegates can then vote and engage in executive votes and governance polls using the entire voting power delegated. A factory contract is provided facilitating the deployment of VoteDelegate contracts.

Two types of VoteDelegate were implemented to be compatible with the existing Maker *DSChief* and Sky *Chief* respectively. The same factory is implemented for both types of VoteDelegate. Note that the two VoteDelegate implementations are only available at distinct commits.

2.2.1 VoteDelegateFactory

This factory contract facilitates the deployment of individual VoteDelegate contracts. For each address (delegate) one VoteDelegate contract can be deployed at a deterministic address. The factory keeps track of the deployed contracts.

Upon deployment, the immutable addresses for the `chief` and `polling` contracts are initialized.

The following public functions are provided:

- `create()` - Permissionless function to deploy a VoteDelegate contract for oneself as a delegate. The contract is initialized with the `chief` and `polling` provided by the factory contract.

The following public view functions are provided:

- `getAddress()` - Based on the given address, calculates the address of the VoteDelegate contract.
- `isDelegate()` - Returns `true` if there is a VoteDelegate contract deployed for the given address.
- `delegates()` - Returns the address of the VoteDelegate contract of the given delegate or 0x0 if such a contract doesn't exist. This function is backward compatible with the previous version.

The contract further exposes public getter functions to access the state: To access the immutable address set upon deployment `chief` and `polling`.

In the lastest version of the factory (last changes in **Version 4**), The factory deploys the contracts with `CREATE` instead of `CREATE2`. Note that `getAddress()` is not available anymore. Otherwise, interaction remains the same.

2.2.2 Maker VoteDelegate

The VoteDelegate contract integrates with the DSChief to participate in the continuous approval voting process.

Upon deployment the immutable addresses for the `chief` and `polling` contracts, as well as the address of the `delegate` and `gov` (the governance token (queried from `chief.GOV()`) are set.

The following public functions are present:

- `lock()` - Allows anyone to lock the specified amount of governance tokens in this VoteDelegate contract. Technically tokens are transferred from the caller to the VoteDelegate contract before being locked in the chief. The VoteDelegate contract maintains an internal accounting of the depositor. Emits the `Lock` event.
- `free()` - Allows to free a specified amount of previously locked governance tokens. Ensures the caller has sufficient stake before freeing the corresponding amount of governance tokens at the chief and transferring them onward to the caller. Emits the `Free` event.

Permissioned voting functions callable only by the respective delegate of this VoteDelegate instance:

- `vote(address[] memory yays)` - calls the respective function on the chief contract, saves a set of ordered candidates (identified as slate) and votes for this slate.
- `vote(bytes32 slate)` - execute a vote on the chief contract for a specific slate (set of candidates).
- `votePoll(uint256 pollId, uint256 optionId)` - execute an indicative poll on the polling contract.
- `votePoll(uint256[] calldata pollIds, uint256[] calldata optionIds)` - execute multiple indicative polls on the pooling contract.

The contract further exposes public getter functions to access the state: To access entries of the mapping `stake` and the immutables addresses set upon deployment `delegate`, `gov` (the governance token), `chief` and `polling`.

2.2.2.1 Changes in Version 2

In `VoteDelegatee.free()` the call to `chief.free()` may revert if a previous call to lock tokens has been made by the same VoteDelegatee in the same block. Some integrators or users require successful unlocking to be ensured within a reasonable timeframe even under adversarial circumstances. A hatch has been introduced allowing to reserve a time slot for successful unlocking.

Anyone may call `reserveHatch()` to reserve an unlock slot starting at the next block and spanning over 5 blocks (hardcoded `HATCH_SIZE`). During this time calls to `VoteDelegatee.lock()` are inhibited, guaranteeing calls to `VoteDelegatee.free()` to succeed. After the hatch ends, a cooldown period of 20 blocks (hardcoded `HATCH_COOLDOWN`) is enforced before the reserve hatch can be triggered again.

2.2.2.2 Changes in Version 3

`reserveHatch()` has been updated to allow an unlock slot to be reserved at the end of the last block of the cooldown period, rather than waiting for the block after the cooldown period.

2.2.3 Sky VoteDelegate

The Sky VoteDelegate shares similar functionalities as Maker VoteDelegate while it is intended to integrate with the Sky *Chief* contract, which:

- allows `lock()` and `free()` in one block if there is no lift.
- prevents `free()` if there is a lift in the same block.
- has a global `liftCooldown` period to preserve blocks between lifts for funds freeing.
- uses SKY as the `gov` token and has no IOU token.

Upon deployment the immutable addresses for the `chief` and `polling` contracts, as well as the address of the `delegate` and `gov`, the governance token (queried from `chief.gov()`), are set.

There is no approval of IOU token anymore since it is removed from the Sky Chief.

Similar to Maker VoteDelegate, the following public functions are present:

- `lock()` - Allows anyone to lock the specified amount of governance tokens in this VoteDelegate contract. Technically tokens are transferred from the caller to the VoteDelegate contract before being locked in the chief. The VoteDelegate contract maintains an internal accounting of the depositor. Emits the `Lock` event.
- `free()` - Allows to free a specified amount of previously locked governance tokens. Ensures the caller has sufficient stake before freeing the corresponding amount of governance tokens at the chief and transferring them onward to the caller. Emits the `Free` event.

Since Sky Chief implements a liftCooldown mechanism to avoid DoS attacks with frequent lift() calls, the hatch mechanism is not needed in the Sky VoteDelegate.

Permissioned voting functions callable only by the respective delegate of this VoteDelegate instance:

- `vote(address[] memory yays)` - calls the respective function on the chief contract, saves a set of ordered candidates (identified as slate) and votes for this slate.
- `vote(bytes32 slate)` - execute a vote on the chief contract for a specific slate (set of candidates).
- `votePoll(uint256 pollId, uint256 optionId)` - execute an indicative poll on the polling contract.
- `votePoll(uint256[] calldata pollIds, uint256[] calldata optionIds)` - execute multiple indicative polls on the pooling contract.

The contract further exposes public getter functions to access the state: To access entries of the mapping `stake` and the immutables addresses set upon deployment `delegate`, `gov` (the governance token), `chief` and `polling`.

2.3 Trust Model

Maker VoteDelegate and Sky VoteDelegate have the following intersections of the trust model:

Delegate: The delegate can use their VoteDelegate contract to vote leveraging the voting power of the delegators. Users who delegate their voting power need to trust their designated delegate. The delegate is unable to access the users' governance tokens.

Polling: Untrusted. Simple contract called to emit events. Used to measure the sentiment of voters.

2.3.1 Maker VoteDelegate

Integrators and users must be aware that `VoteDelegatee.free()` could fail, occasionally under normal operation or repeatedly under adversarial circumstances. They should be prepared to use the reserve hatch functionality if necessary. For this, they are expected to be able to complete their unlocking transaction within five blocks of calling `reserveHatch()`. Moreover, the integrating protocol must be able to handle a delay of a maximum hatch cooldown period (20 blocks + at minimum 1 to unlock), during which triggering the hatch is not possible.

Governance token: Assumed to be the MKR token. Fully trusted to work correctly. This token must revert upon failed transfers as the code under review ignores return values.

Chief: Assumed to be Maker *DSChief*; fully trusted. Recipient of the tokens to be locked. Must work correctly as expected or funds could be lost. Freeing tokens in the VoteDelegate contract relies on this contract returning the tokens.

2.3.2 Sky VoteDelegate

Integrators and users must be aware that `VoteDelegatee.free()` could still fail if there is a prior `Chief.lift()` call in the same block. Though freeing funds is feasible in the `liftCooldown` period.

Governance token: Assumed to be the SKY token, a rebranded version of MKR. Fully trusted to work correctly. This token must revert upon failed transfers as the code under review ignores return values.

Chief: Assumed to be Sky *Chief*; fully trusted. Recipient of the tokens to be locked. Must work correctly as expected or funds could be lost. Freeing tokens in the VoteDelegate contract relies on this contract returning the tokens.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe our findings. The findings are split into these different categories:

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

6.1 Flashloan Protection

Note **Version 1**

Both chief implementations feature a flashloan protection mechanism that blocks `free()` from unlocking tokens.

- *Maker VoteDelegate*: DSChief prevents calling `free()` if the caller has previously executed `lock()` in the same block. Hence, when a user locks tokens in a VoteDelegate contract, any subsequent attempts by other users of the same VoteDelegate to execute `free()` within the same block will fail. In Version 2 a hatch functionality to reserve a time slot for unlocking has been introduced (refer to the System overview for a detailed description). While the hatch is active, transactions to `lock()` will fail.
- *Sky VoteDelegate*: The new chief version prevents calling `free()` if `lift()` has been previously executed in the same block. Thus, lifting a new `hat` can block withdrawals. However, the chief provides a cooldown period where users will be able to withdraw.