CHAINSECURITY

PUBLIC

Code Assessment
of the RewardsBeam Smart Contracts

November 05, 2026

Produced for

Sky



Contents

1	Executive Summary	3
2	Assessment Overview	4
3	System Overview	5
4	Trust Model	7
5	System Considerations	8
6	Terminology	9
7	Findings	10
8	Limitations and use of report	11

1 Executive Summary

Dear Sky Team,

Thank you for trusting ChainSecurity with the review of RewardsBeam. As a globally trusted security partner for decentralized finance protocols and their governance, ChainSecurity provides specialized, independent technical analysis of complex smart contract systems. We greatly appreciate the opportunity to collaborate with Sky on this project.

RewardsBeam lets a facilitator change the reward rate of a treasury-funded farm without a governance spell, within limits set by Sky governance. `RewardsBeamMom` lets governance stop it without the governance delay.

We have conducted a thorough review of the scope. The most important areas covered in our review are:

- The limits on the facilitator, since it can change the rewards without the governance delay.
- The integration with `DssVest` and `VestedRewardsDistribution`, since every `set()` replaces the stream of the farm.
- The funding of the streams, since the vest is shared with other farms. We highlight that beam streams use the same treasury allowance as the other streams, see **Beam Streams Share the Vest Allowance with Other Farms**.

We provide this private audit report to assist you in forming an opinion on the risk profile of RewardsBeam. In summary, we find that the codebase provides a high level of security, if operated under the assumptions documented in **System Overview**, **Trust Model** and **System Considerations**.

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 and our observations. We are happy to receive questions and feedback to improve our service.

Sincerely yours,
ChainSecurity

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 RewardsBeam repository based on the documentation files. The code versions relevant to this report and when they were received are indicated below.

Version	Date	Commit Hash	Note
1	2 November 2026	cca3b72677999ffa81937a88a6755853741e3111	Initial review

For the Solidity smart contracts, the compiler version `0.8.24` and the EVM version `cancun` were chosen.

The following files were in scope:

```
deploy/
├── RewardsBeamDeploy.sol
├── RewardsBeamInit.sol
└── RewardsBeamInstance.sol
src/
├── RewardsBeam.sol
└── RewardsBeamMom.sol
```

2.1.1 Excluded from scope

All other files, including tests and external libraries, are not part of this assessment. The contracts the system interacts with, such as `DssVest`, `VestedRewardsDistribution` and `StakingRewards`, are not in scope either. They are assumed to behave correctly and according to their specification.

3 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Sky funds farms from its treasury. A `DssVest` stream pays the rewards to a `VestedRewardsDistribution` contract, which forwards them to the farm (`StakingRewards`). So far, changing the reward rate of such a farm required a governance spell (`TreasuryFundedFarmingInit.updateFarmVest()`).

RewardsBeam lets a facilitator change the reward rate of one farm without a spell, within limits set by governance. `RewardsBeamMom` lets governance stop it without the governance delay.

3.1 RewardsBeam

`RewardsBeam` is linked to one `VestedRewardsDistribution` (`dist`) and its `DssVest` (`vest`). Following the usual Sky conventions, two roles manage it:

- **Wards** add and remove wards (`rely()`, `deny()`) and facilitators (`kiss()`, `diss()`), and set the parameters with `file()`.
- **Facilitators** (buds) call `set(vestTot)`.

The parameters are:

- `maxVestTot`: the maximum total amount of a new stream.
- `vestTau`: the duration of a new stream, at most 20 years.
- `tau`: the cooldown between two `set()` calls.
- `toc`: the time of the last `set()` call. `set()` updates it.

`set(vestTot)` replaces the current stream of `dist` with a new one. It reverts if the cooldown has not passed or if `vestTot` is above `maxVestTot`. Otherwise, it:

1. calls `dist.distribute()` if the current stream has unpaid rewards, so that the farm receives them,
2. yanks the current stream,
3. creates a new restricted stream of `vestTot` to `dist`, which starts now and lasts `vestTau`,
4. sets the new stream in `dist`.

Hence, the reward rate is at most `maxVestTot / vestTau`. The `cap` of the vest also limits it. `set()` cannot create the first stream, so governance must set one in `dist` first. The beam does not change the allowance of the treasury to the vest, see Beam Streams Share the Vest Allowance with Other Farms.

3.2 RewardsBeamMom

`RewardsBeamMom` has an owner (the pause proxy) and an authority (the chief). `stop()` can be called by the owner and by the addresses the authority allows, i.e., the hat. It sets `tau` of the beam to `type(uint64).max`, so that `set()` can no longer be called, and yanks the current stream. Rewards earned until then can still be distributed. Governance can enable the beam again by setting `tau` back.

3.3 Deployment

`RewardsBeamDeploy` deploys the beam and the mom and transfers their ownership to the given owner. Then, the pause proxy executes `RewardsBeamInit` in a spell. It:

- relies the beam on `vest` and `dist`, and the mom on the beam and `vest`,
- optionally sets the allowance of the treasury to `vest`,
- sets the parameters and the facilitators of the beam,
- sets the chief as the authority of the mom,
- adds both contracts to the chainlog.

4 Trust Model

4.1 Roles

Sky Governance (pause proxy)

- Trust level: Fully trusted.
- Permissions: Ward of the beam and owner of the mom. Ward of `DssVest` and `VestedRewardsDistribution`. Sets all parameters, adds and removes facilitators, and funds the streams from the treasury.
- Assumptions: Governance sets correct parameters. It keeps the allowance and the balance of the treasury high enough for all streams of the vest, see Beam Streams Share the Vest Allowance with Other Farms. When it yanks the stream of the farm outside of the beam, it also disables the beam.
- Worst-case impact: Arbitrary changes to the rewards of the farm and to all other streams of the vest.

Chief hat

- Trust level: Fully trusted.
- Permissions: Call `stop()` on the mom without the governance delay.
- Worst-case impact: Stop the rewards of the farm until governance enables the beam again.

Facilitators

- Trust level: Partially trusted. Expected to be a multisig.
- Permissions: Call `set()` once per cooldown, with a total of at most `maxVestTot`.
- Worst-case impact: Set the reward rate of the farm to any value up to `maxVestTot / vestTau`, including almost zero, and restart the stream after every cooldown. Use allowance of the treasury that other streams of the vest also need.

Users

- Trust level: Untrusted.
- Permissions: None on the beam and the mom. They can call the permissionless functions of the integrated contracts at any time, e.g., `distribute()` of `VestedRewardsDistribution`.

4.2 External Dependencies

`DssVest`, `VestedRewardsDistribution`, `StakingRewards` and the `Chief` are assumed to be the audited Sky implementations. The vest is assumed to be a `DssVestTransferrable` that pays from the treasury. It is shared with other farms, and the beam and the mom are its wards. The reward token is assumed to be a standard ERC-20 token without fees or callbacks, e.g., SKY.

5 System Considerations

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

SC1 `RewardsBeam.set()` Can Revert Due to the Vest Cap

System Consideration **Version 1**

`DssVest.create()` checks that the rate of a new stream is not above the `cap` of the vest (`tot / tau <= cap`). Since `RewardsBeam.set()` creates a new stream, it reverts with `DssVest/rate-too-high` if `vestTot / vestTau` is above `cap`.

The beam does not check its parameters against `cap`. If `maxVestTot / vestTau` is above `cap`, `set()` reverts for some values of `vestTot` that `maxVestTot` allows.

Further, `cap` is shared by all streams of the vest and can change at any time, e.g., in a spell for another farm. So the limit of the beam can change without any change to the beam. The `README.md` mentions `cap` as a second rate limit that `vest.create()` enforces.

SC2 Beam Streams Share the Vest Allowance with Other Farms

System Consideration **Version 1**

A `DssVestTransferrable` (e.g., `MCD_VEST_SKY_TREASURY`) pays every stream with `transferFrom(czar, ...)`. So all its streams use one allowance from the treasury (`MCD_PAUSE_PROXY`) to the vest. Other farms already stream from the same vest.

`RewardsBeam.set()` creates new streams, but it never changes this allowance. Every token that a beam stream pays reduces the allowance that the other streams also need. Governance should be aware that a facilitator can, within its boundaries, use allowance that governance planned for another farm. If the allowance becomes too low, `distribute()` of the other farm reverts until governance increases the allowance.

In comparison, the spell routine `TreasuryFundedFarmingInit.updateFarmVest()` adds the new stream to the current allowance.

This behavior is documented in the `README.md` and in a comment in `RewardsBeamInit.init()`: keeping the allowance high enough for the beam and the other streams is governance's job.

Further, `TreasuryFundedFarmingInit.updateFarmVest()` assumes that the allowance covers the remaining amounts of the streams. Hence, spells that use it for a farm of the vest can revert, e.g., if a facilitator calls `set()` on the same farm during the governance delay.

6 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

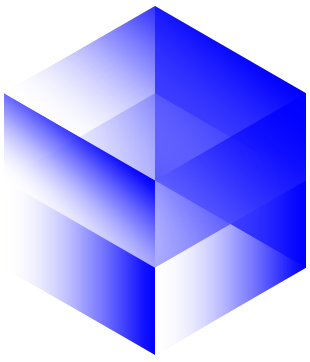
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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.

7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



No Open Issues

Total number of findings

Critical	High	Medium	Low	Informational
0	0	0	0	0

8 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.