PUBLIC

Code Assessment

of the DSS Emergency Spells Smart Contracts

April 17, 2025

Produced for

by

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 DSS Emergency Spells according to Scope to support you in forming an opinion on their security risks.

Sky implements DssEmergencySpells, a set of pre-defined emergency spells that bypass the governance delay defined in DSPause to enable prompt governance actions if necessary.

The most critical subject covered in our audit is functional correctness. After the intermediate report, all findings have been resolved, hence security regarding functional correctness is high.

The general subjects covered are trustworthiness and documentation. 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	2
• Code Corrected	2

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 DSS Emergency Spells repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	19 Sep 2024	bc06eebcec04800f778b0bef339f5253222209d2	Initial Version
2	24 Oct 2024	1feca1d3feef19783944c8b86142c7d19f8cddfd	After Intermediate Report
3	21 Nov 2024	f9308604f731ab180f6eb10d463b3edeb6a9a127	Rename LineWipe Spell
4	09 Dec 2024	0129bcd1e7b5a91782aceb5b5d2c5ece5e4138ff	Additional Spells
5	13 Dec 2024	107a165afa8d801df424c179666f762f629eae49	Natspec Description Corrected
6	13 Feb 2025	a052bb416fae2e03b64756c87eb67ec1a1a818a1	Amended Broken License Headers
7	11 Apr 2025	fd1c76e644e845b349f3b7072666db60b1602573	SPBEAM Halt Spell
8	16 Apr 2025	ea10a1b2305395c7885c134d4d79105af47256bb	README update

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

The following contracts were in scope:

```
src/
    DssEmergencySpell.sol
    osm-stop/
        SingleOsmStopSpell.sol
        MultiOsmStopSpell.sol
    clip-breaker/
        SingleClipBreakerSpell.sol
        MultiClipBreakerSpell.sol
    auto-line-wipe/
        SingleAutoLineWipeSpell.sol
        MultiAutoLineWipeSpell.sol
    ddm-disable/
        SingleDdmDisableSpell.sol
```

In Version 3, the following files:

```
src/
    auto-line-wipe/
        SingleAutoLineWipeSpell.sol
        MultiAutoLineWipeSpell.sol
```

were renamed to:

```
src/
    line-wipe/
        SingleLineWipeSpell.sol
        MultiLineWipeSpell.sol
```

In Version 4, the following files were added to scope:

```
src/
    DssGroupedEmergencyspell.sol
    splitter-stop/
        SplitterStopSpell.sol
    lite-psm-halt/
        SingleLitePsmHaltSpell.sol
    clip-breaker/
        GroupedClipBreakerSpell.sol
    line-wipe/
        GroupedLineWipeSpell.sol
```

In Version 7, the following file was added to scope:

```
src/
    spbeam-halt/
        SPBEAMHaltSpell.sol
```

2.1.1 Excluded from scope

Generally, all files not mentioned above are out of scope.

2.2 System Overview

This system overview describes 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 offers a set of pre-defined emergency spells that bypass the governance delay defined in DSPause.

With continuous approval voting in `MCD_ADM` governance token holders can elect a `hat` which has the authorization to call privileged functions in the system. More specifically, the `hat` can call functions access-controlled by the `canCall` function of `MCD_ADM`. That typically allows regular spells to register an action for execution in DSPause so that it can be executed once the governance delay passes. In contrast, emergency spells bypass the governance delay by not interacting with DSPause but with other contracts that define access control with `canCall` function of `MCD_ADM` (e.g. certain functionality in mom contracts).

Thus, a suite of contracts implementing pre-defined emergency actions is provided to allow for quickly taking action if necessary.

2.2.1 General

The generic and abstract emergency spell contract DssEmergencySpell implements the shared logic for all emergency spells. For compatibility with existing tooling, the emergency spells share the interface with the regular ones (see DssExec). Given the lack of an action, the interface of actions (see DssAction) is also implemented.

The core functionality is abstract and consists of:

- `description()`: *Abstract*. An accurate description for the spell (e.g. suitable name).
- `done()`: *Abstract*. Returning `false` defines that the spell could be executed (if the spell is the `hat`). Returning `true` indicates that no action needs to or can be taken (e.g. completed, misconfiguration, or similar). Note that the spells do not have state and are reusable. Thus, this could become `true` or `false` due to external factors not related to the emergency spell.
- `schedule()`: Performs the emergency actions with the `abstract` function `_emergencyActions()`. Note that `schedule()` is the first action taken on a regular spell. Thus, for compatibility reasons with existing tooling, the execution is implemented in this function.

The remaining functionality is purely present for interface compatibility:

- `action()`: Returns `address(this)` since no action contract exists for emergency spells. However, given that the action interface is implemented, the return value is somewhat meaningful.
- `cast()`: No-Op. For emergency spells, the function is not expected to perform any operation.
- `eta()`: Zero. `eta` is used to enforce the GSM delay between the `schedule()` and `cast()`. While it is zero, the spell can be scheduled. Thus, zero is chosen.
- `expiration()`: `uint256.max`. The pre-defined emergency spells are never expected to expire.
- `log()`: Returns the address of the Chainlog contract.
- `nextCastTime()` / `nextCastTime(uint256)`: `uint256.max` since emergency spells are never expected to be cast.
- `officeHours()`: `false`. Office hours do not apply to emergency spells.
- `pause()`: Returns the `MCD_PAUSE` contract.
- `sig()`: Returns the selector of `execute()`. Note that this is the typical value and that the selector is supported by emergency spells.
- `tag()`: Returns the codehash of `address(this)` as it is the address of the action.
- `execute()`: No-Op. For emergency spells, the function is not expected to perform any operation.
- `actions()`: No-Op. For emergency spells, the function is not expected to perform any operation.

Further note that grouped emergency spells are defined, enabling an emergency action to be applied to multiple ilks set in the constructor. The abstract DssGroupedEmergencySpellLike extends DssEmergencySpellLike and implements:

- `ilks()`: Returns the list of ilks to which the spell applies.
- `description()`: Returns the spell description: The full description is formed by concatenating the description prefix with a comma-separated list of ilks.
- `emergencyActionsInBatch()`: Entrypoint to execute the emergency actions for selected ilks (between `start` and `end` of the `ilkList`) in a batch.
- `done()`: Returns whether the spell is done for all ilks of the group or not.

The following functionalities must be implemented by the contract extending the abstract contract:

- `_descriptionPrefix()`: Returns the description prefix used to build the final description with the appended ilks.
- `_emergencyActions(bytes32 _ilk)`: Implements the emergency actions for a specified ilk.
- `_done()`: Indicates whether the spell can be executed. See the General section for details.

2.2.2 Emergency Spells

This section describes the individual emergency spells.

Note that for some spell types, there is a single-ilk and a multi-ilk version of the spell. The former can be permissionlessly deployed by a factory and is performing the action solely for one ilk which is specified on deployment. The latter iterates over the ilks registered in the ilk registry. Typically, for the multi-ilk version an additional function to iterate over a subset of ilks is provided. Additionally, for some spell types grouped emergency spells are provided. Factories are typically provided to deploy such spells.

Line Wipe: Both a single- and a multi-ilk version are implemented. Additionally, a grouped version is provided. The spells disable generating further debt for ilks by calling `wipe()` on the LineMom. That clears the AutoLine storage for an ilk and sets the ilk's `line` in the Vat to zero. A spell is considered done if the ilks' AutoLine storage and the ilks' Vat line are zero.

Clip Breaker: Both a single- and a multi-ilk version are implemented. Additionally, a grouped version is provided. The spells disable collateral auctions by calling the function `setBreaker()` on the ClipperMom for the ilks' Clippers (e.g. Clipper, LockstakeClipper) to set the breaker level to 3 which disables the functions `kick()`, `redo()` and `take()`. A spell is considered done if the breaker level on the ilks' Clippers is set to 3.

DDM Disable: Only a single-ilk version is implemented. The spell disables the D3MPlan (e.g. D3MOperatorPlan) for an ilk by calling disable on the D3MMom for the ilk's D3MPlan. That effectively makes `active()` return `false` so that the D3MHub only unwinds ilk's D3MPool's position. A spell is considered done if the ilk's plan's `active()` function returns `false`.

OSM Stop: Both a single- and a multi-ilk version are implemented. Stops the ilks' OSM (e.g. OSM, CurveLPOracle) by calling `stop()` on the OsmMom. That disables an ilk's respective oracle by calling `stop()` on it. A spell is considered done if the ilks' OSM are stopped.

SingleLitePsmHaltSpell: Only a single psm version is implemented. The spell halts certain flows (`SELL`, `BUY`, or `BOTH`) of LitePsm by calling the function `halt()` on the LitePsmMom to set the swap fees `tin` or `tout` to `HALTED` (`type(uint256).max`). A spell is considered done if the swap fees for respective flows are set to `HALTED`.

SplitterStopSpell: The spell stops the Splitter by calling the function `stop()` on the SplitterMom to set the `hop` (the cooldown period before the next `kick`) to `type(uint256).max`. A spell is considered done if the `hop` is `type(uint256).max`.

SPBEAMHaltSpell: The spell stops the SPBEAM by calling the function `halt()` on the SPBEAMMom to set the `bad` (the circuit breaker flag) to `1`, preventing future rate updates with SPBEAM. A spell is considered done if the `bad` is `1`.

2.2.3 Changelog

In **Version 4**, the emergency spells for halting LitePsm and Splitter were added. Additionally the grouped line wipe and the grouped Clip breaker have been added. Note that in this version grouped spells were first introduced.

In **Version 7**, the emergency spell for halting SPBEAM was added.

2.2.4 Roles & Trust Model

Emergency spells are immutable contracts with no special privileges by default. There are no privileged roles in the contracts and any functionality is permissionless.

Should the need arise, governance token holders can elect an emergency spell to be the `hat` in `MCD_ADM`, giving it certain privileges to execute the predefined actions with `schedule`.

The majority of governance token holders is trusted to act honestly and correctly at all times and to vote for such a proposal when needed.

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 any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Correctness: Mismatches between specification and implementation

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 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

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	2

- Incorrect Use of try/catch Code Corrected
- No Vat line Check for Line Wipes Code Corrected

Informational Findings	1

- Incorrect Parameter Description Specification Changed

6.1 Incorrect Use of `try/catch`

Correctness Low Version 1 Code Corrected

CS-MKR-ES-001

In the multi-ilk version spells (MultiClipBreakerSpell and MultiOsmStopSpell), the `try/catch` pattern is used during the spell execution. It is intended to skip an incompatible contract, which does not implement the typical interfaces.

However, the catch clause used is `catch Error(string memory reason)`. When a function call is made to a contract that does not implement the expected interface, the execution reverts with empty error data instead of an error with signature `Error(string)`. Consequently, the catch block will fail to handle such cases, causing the entire function to revert and the emergency spell to fail.

Code corrected:

Spells have been corrected to use clause `catch (bytes memory reason)` for any other type of reverts in the external call.

6.2 No Vat `line` Check for Line Wipes

Correctness Low Version 1 Code Corrected

CS-MKR-ES-002

The `done()` functions will generally only return `true` if zeroed state is returned by the AutoLine contract. However, `Vat::line()` is not checked.

Consider the following scenario:

1. The ilk is registered and used. An AutoLine configuration is set up.
2. Eventually, the configuration in AutoLine is temporarily disabled and hence deleted. Thus, the ilk remains registered in the LineMom.

3. Now, `done()` could return `true` even if the `line` has not been fully wiped (LineMom wipes the `line`, too).
4. As a consequence, a call to `wipe()` might be missed.

Ultimately, the `Vat::line()` could be non-zero, leading to scenarios where properly set up ilks can generate debt.

Code corrected:

Spells have been corrected to take `Vat::line()` into consideration.

6.3 Incorrect Parameter Description

Informational | Version 1 | Specification Changed

CS-MKR-ES-003

The natspec description of `DssGroupedEmergencySpell._emergencyActions()` reads:

```
@param _ilk The ilk to set the related Clip breaker.
```

This description is specific to the GroupedClipBreakerSpell, in general for the abstract DssGroupedEmergencySpell this is the ilk for which the emergency action is executed.

Specification changed:

Specification has been corrected to align with the contract.

7 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.

7.1 Considerations for Clipper Breaker

Note Version 1

The Clipper Breaker spells disable clipper functions for a set of ilks. Note the following:

1. The multi-ilk spell will only operate on ilks that have a non-zero clipper registered in the ilk registry.
2. The multi-ilk spell will only operate on ilks where the ClipperMom is the ward of the respective Clipper.
3. The multi-ilk spell will only operate on ilks if the functions with the respective arguments are supported on the Clipper. Thus, to ensure consistency between `done()` and `schedule()` the property should hold that if `stopped()` is supported by the Clipper, the Clipper should support `file("stopped", 3)` calls from the mom, too, and vice versa.

Note that there is a subtle difference between the multi- and the single-ilk spells:

1. The `schedule()` function of the single-ilk spell could revert for an ilk (e.g. if the mom is not authorized on the Clipper). In contrast, the same function in the multi-ilk spell will skip such ilks.

7.2 Considerations for DDM Disable

Note Version 1

The DDM Disable spells disable an ilk's D3M plan so that only unwinding can happen. Note the following:

1. It is assumed that the D3MMom is authorized in the plan.

7.3 Considerations for Line Wipe

Note Version 1

The Line Wipe spells disable generating more debt for a set of ilks. Note the following:

1. The multi-ilk spell will only operate on ilks that have also been registered in the LineMom and will skip the execution and done-check for any other ilk registered in the ilk registry.
2. It is assumed that no other mechanism besides the governance's Vat interactions and the AutoLine can adjust `line`.
3. It is assumed that the LineMom is authorized in the AutoLine and the Vat.

Note that there is a subtle difference between the multi- and the single-ilk spells:

1. The `schedule()` function of the single-ilk spell will revert for an ilk if it is not registered in the mom contract. In contrast, the same function in the multi-ilk spell will skip such ilks during the iteration.

7.4 Considerations for OSM Stop

Note Version 1

The OSM Stop spells stop the OSMs for a set of ilks. Note the following:

1. The multi-ilk spell will only operate on ilks that have also been registered in the OsmMom and will skip the execution and done-check for any other ilk registered in the ilk registry.
2. The multi-ilk spell will only operate on ilks where the OsmMom is the ward of the respective oracle.
3. The multi-ilk spell will only operate on ilks if the functions with the respective arguments are supported on the oracle. Thus, to ensure consistency between `done()` and `schedule()` the property should hold that if `stopped()` is supported by the OSM, the OSM should support `stop()` calls from the mom, too, and vice versa.

Note that there is a subtle difference between the multi- and the single-ilk spells:

1. The `schedule()` function of the single-ilk spell could revert for an ilk (e.g. if the mom is not authorized on the oracle). In contrast, the same function in the multi-ilk spell will skip such ilks.

7.5 Out-of-Gas for Multi-Ilk Spells

Note Version 1

Governance should be aware that for multi-ilk spells the `schedule()` and the `done()` function could run out-of-gas. For `schedule()`, an alternative function is typically offered that allows iterating over a subset. `done()` is intended for off-chain use.

Further, note that the calls in `try` / `catch` might run out-of-gas, potentially leading to a scenario where the last call might fail but the overall execution succeeds (even though with sufficient gas it would have succeeded). However, `done()` should indicate off-chain that the spell has not been completed.

7.6 Single-Ilk Spell Factories

Note Version 1

Governance should be aware that:

1. Multiple single-ilk spell contracts could be deployed by the factory for the same ilk. That might require additional coordination efforts when voting for executing a single-ilk spell.
2. For non-existing ilks, a single-ilk spell could exist, too.