

# Code Assessment of the Dss Flappers Smart Contracts

July 10, 2026

Produced for



by



# Contents

|          |                                      |           |
|----------|--------------------------------------|-----------|
| <b>1</b> | <b>Executive Summary</b>             | <b>3</b>  |
| <b>2</b> | <b>Assessment Overview</b>           | <b>5</b>  |
| <b>3</b> | <b>Limitations and use of report</b> | <b>13</b> |
| <b>4</b> | <b>Terminology</b>                   | <b>14</b> |
| <b>5</b> | <b>Open Findings</b>                 | <b>15</b> |
| <b>6</b> | <b>Resolved Findings</b>             | <b>16</b> |
| <b>7</b> | <b>Informational</b>                 | <b>20</b> |
| <b>8</b> | <b>Notes</b>                         | <b>22</b> |

# 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 Flappers according to [Scope](#) to support you in forming an opinion on their security risks.

Sky has implemented new contracts to process the surplus of the stablecoin system. A new Splitter contract divides the surplus between a burning engine (Flapper) and a reward farm. Flapper contracts interact with UniswapV2, exchanging USDS for Gem tokens, with two variants: FlapperUniV2SwapOnly fully converts USDS to Gem, while FlapperUniV2 adds liquidity to the pool. A Kicker contract has been implemented which provides a new entrypoint to initiate surplus processing.

The latest version introduces SBEBeam, which allows facilitators to configure the splitter, kicker, and farms within governance-defined bounds. To let both governance and SBEBeam interact with the farms, a farm owner contract has been implemented that invokes owner functionality on a given farm on behalf of these auth addresses.

The most critical subjects covered in our audit are access control, asset solvency, functional correctness, and the impact of the change on the existing system. An issue with functional correctness was identified, where `Splitter.cage()` did not lock the Splitter completely (see [Splitter.cage\(\) does not lock the Splitter](#)). After the intermediate report, this issue has been resolved.

The general subjects covered are specifications correctness, optimizations, and soundness of the deployment and initialization scripts. The specification of `babylonian.sqrt()` was inaccurate (see [Incorrect specification](#)). The checks in the initialization scripts could be further enhanced (see [Missing check for bump](#) and [Missing check of reward token on farm contract](#)). All the issues have been resolved and 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.

|                                    |   |
|------------------------------------|---|
| <b>Critical</b> -Severity Findings | 0 |
| <b>High</b> -Severity Findings     | 0 |
| <b>Medium</b> -Severity Findings   | 0 |
| <b>Low</b> -Severity Findings      | 6 |
| • <b>Code Corrected</b>            | 6 |

## 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 Flappers 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 | 29 Jan 2024 | <a href="#">4e7a8f452a463a243f3bb2ef065cf5f517af6490</a> | Initial Version                 |
| 2 | 07 Feb 2024 | <a href="#">f039433bc80c0e5009cf7bf3fc2008e12f9c1c46</a> | After Intermediate Report       |
| 3 | 09 May 2024 | <a href="#">bf96463d5a3ec4ae82845668c43e57311d3aa3dc</a> | Farm Mutable & Compiler Version |
| 4 | 28 Aug 2024 | <a href="#">95431f3d4da66babf81c6e1138bd05f5ddc5e516</a> | Usage of USDS                   |
| 5 | 17 Oct 2025 | <a href="#">f4f4f22b3eae6c912551b00ad64a56862ad61f86</a> | Kicker                          |
| 6 | 06 Jul 2026 | <a href="#">9f47925396bd7ee5885f8c30678a142b7783a407</a> | Beam                            |
| 7 | 09 Jul 2026 | <a href="#">655d2dd2c7000235633fec87b763ee0143e85bab</a> | Beam Fixes                      |

For the solidity smart contracts, the compiler version 0.8.16 was chosen. In Version 3, the compiler version was changed to 0.8.21.

The scope consists of the following Solidity files:

```
./src/FlapperUniV2.sol
./src/FlapperUniV2SwapOnly.sol
./src/OracleWrapper.sol
./src/Splitter.sol
./src/SplitterMom.sol
./src/Babylonian.sol

./deploy/FlapperDeploy.sol
./deploy/FlapperInit.sol
./deploy/SplitterInstance.sol
```

Since **Version 5**, the following file has been added to the scope:

```
./src/Kicker.sol
```

Since **Version 6**, the following files are in scope:

```
./src/SBEBeam.sol  
./src/FarmOwner.sol
```

For the deployment scripts, the review covered the following:

- Deployment: Confirmed proper deployment mechanisms, including accurate parameter relay from script to constructor and, if required, ownership transitions.
  - Initialization: This code is to be executed as delegate call in the context of the Governance pause proxy. It is assumed the Splitter, Flapper, and Oracle / OracleWrapper are initialized atomically.
1. Ensuring the code is performing thorough sanity checks on already configured state variables (except mappings, which is not feasible).
  2. Sanity checks and initialization of further states.
  3. Distributing essential access rights.
  4. Updating the chainlog.

Previous reports have covered earlier versions of the Flapper implementations. This report focuses exclusively on the current implementation as outlined above.

### 2.1.1 Excluded from scope

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

UniswapV2 is not in scope of this review.

The selection of the parameters (e.g. bump, kbump, khump) is out of scope of this review.

## 2.2 System Overview

This system overview describes the latest version (**Version 7**) of the contracts as defined in the [Assessment Overview](#).

Sky implemented new contracts to process the surplus of the stablecoin system. A Splitter contract is introduced to split the surplus between the Flapper and the USDS reward farm contract. Two versions of the Flapper implementation exist: FlapperUniV2 and FlapperUniV2SwapOnly. In Version 5, a new Kicker contract is introduced to replace the native Vow.flap() functionality. In Version 6, the SBEBeam contract is introduced, allowing permissioned facilitators to update the surplus processing parameters within bounds set by the governance, along with the FarmOwner helper contract, needed as the reward farm only supports a single owner while both the governance and the SBEBeam must administrate it.

Leveraging UniswapV2, USDS is exchanged for Gem tokens. FlapperUniV2SwapOnly converts all USDS and the resulting Gem tokens are transferred to the predefined receiver while FlapperUniV2 converts a partial amount and deposits both USDS and Gem into the liquidity pool with the liquidity pool shares being minted to the predefined receiver.

### Kicker

Introduced in (**Version 5**) of this codebase, the Kicker contract provides a new entrypoint to initiate surplus processing. This is intended to replace Vow.flap() which will be disabled by setting the Vow parameters accordingly. Contrary to Vow.flap(), it no longer features the check that ensures all system debt is accounted for (the requirement that  $\text{Vat.sin}(\text{Vow}) - \text{Sin} - \text{Ash} == 0$ ).

Upon deployment, the immutable addresses of Vat, Vow, and Splitter are initialized in the constructor.

The Kicker contract features the following state variables modifiable by the privileged wards:

- `kbump` - Fixed lot size [rad] to be kicked per surplus auction.
- `khump` - Flap threshold [rad] representing the surplus buffer. This is an `int256` type, allowing for negative values.

Whenever the Vow has sufficient surplus, anyone may execute `Kicker.flap()`. The function verifies that `Vat.dai(Vow) >= Vat.sin(Vow) + kbump + khump`, then mints `kbump` amount of internal dai balance to the Kicker via `Vat.suck()` (consequently increasing Vow's sin), and finally triggers `Splitter.kick(kbump, 0)` to process the surplus through the Splitter.

## Splitter

Serves as the new Flapper contract to be registered in the Vow.

Upon deployment, the immutable address of the `usdsJoin` is initialized.

The Splitter contract features the following state variables modifiable by the privileged roles:

- `burn` Burn percentage expressed in WAD, 1 WAD = funneling 100% to the burn engine (Flapper contract).
- `hop` minimum cool down period (in seconds) between kicks.
- `flapper` Flapper contract / Burn engine where the Burn percentage part of the surplus USDS is transferred to and `exec()` is invoked.
- `farm` Reward farm that receives the surplus to be allocated to a farm.

Whenever the Vow of the DSS has sufficient surplus, anyone may execute `Splitter.kick()` through `Vow.flap()`. Before calling `Splitter.kick(bump, 0)`, the Vow ensures the debt is zero and that there is sufficient surplus. The Splitter distributes the surplus (USDS tokens) according to the burn percentage configured and executes `Flapper.exec(lot)` and `Farm.notifyRewardAmount(pay)`.

## SplitterMom

This contract is used to bypass the governance delay when disabling the Splitter in an emergency.

Since Sky contracts do not feature fine grained access control, any privileged account (ward) may call any privileged function on the contract. To expose certain privileged functions only, Mom contracts are used: These smart contracts feature code which allows to call certain functions on the target contract only, hence restricting the access.

`SplitterMom` must be given the ward role in the Splitter. It exposes `stop()` which updates the `hop` parameter of the Splitter to `type(uint256).max`. Consequently, executions of `kick()` will be paused until the value has been reset.

`stop()` is a privileged function. Access control is determined through a `DsAuth` scheme: Either the contract itself or the owner can call the function. Furthermore, if an authority contract is set, the authority contract is queried whether to grant access for this `msg.sender()` on this function.

Finally the contract offers functionality for the owner to update the owner (`setOwner()`) and the authority contract (`setAuthority()`).

## FlapperUniV2

The contract's main function `exec()` is expected to be called from the Splitter contract following the transfer of USDS tokens. The available USDS amount is processed as follows:

- Calculates the USDS amount to sell for Gem, in order to be able to supply both USDS and Gem to the pool in a balanced way.
- A sanity check on the exchange rate is done by comparing the expected exchange rate with a reference oracle.
- Eventually swap and mint operations are executed on the pool. The received LP tokens are sent to the receiver.

The receiver will be a PauseProxy controlled by the Governance. LP tokens are internally called "Elixir".

Upon deployment of the contract, immutables including the Gem token, pair (UniswapV2 pool) and the receiver are initialized.

The Flapper contract features the following state variables that can be updated by privileged roles:

- pip the reference price oracle used for the sanity check on the exchange rate of the swap.
- want relative multiplier (in units of WAD) of the reference price to insist on during the price check.

Furthermore, there is a variable `live` which is set to 1 upon deployment and to 0 when the contract is caged during global shutdown.

Shutdown / Global settlement:

In step 1 of the shutdown procedure, the End module will call `Vow.cage()` which in turn calls the Flapper. Since this Flapper holds no assets, no funds must be moved, the contract's `live` status is simply disabled.

LP tokens at the receiver are considered lost for the system in case of Emergency Shutdown due to malicious governance.

Administrative functions to add/remove privileged roles (`rely()` / `deny()`) and to update parameters `file()` exist.

### **FlapperUniV2SwapOnly**

Similar to FlapperUniV2, however it only swaps all USDS to Gem (the recipient is the receiver), does not deposit liquidity into the pool.

### **OracleWrapper**

This contract facilitates the conversion between an MKR/USD oracle and the USD value of the new governance token: SKY. The denomination of the new token is that 1 MKR token is equivalent to 24000 of SKY.

### **FarmOwner**

Introduced in **Version 6** of this codebase, the FarmOwner contract holds the ownership of the Synthetix-style StakingRewards farm configured in the Splitter. The farm features a single-owner access control scheme: With the FarmOwner holding the owner role of the farm, each function of the farm restricted to the owner is exposed as a forwarder function restricted to the privileged wards of the FarmOwner. Hence, multiple accounts (intended to be the governance PauseProxy and the SBEBeam) can administrate the farm. In practice, this allows SKY Governance to access all owner restricted functions of the farm, while the SBEBeam contract only calls `setRewardsDuration()` to keep the farm's reward duration in sync with `Splitter.hop`.

Upon deployment, the immutable address of the farm is initialized in the constructor.

All owner-restricted functions of the farm are exposed as forwarder functions restricted to the wards: `setRewardsDuration()`, `setRewardsDistribution()`, `setPaused()`, `nominateNewOwner()`, and `recoverERC20()`. Additionally, `acceptOwnership()` is exposed, allowing the FarmOwner to accept an ownership nomination. As the farm transfers recovered tokens to its owner, `recoverERC20()` additionally forwards the received tokens to a specified recipient within the same call. The recovered token is assumed to revert on failed transfers and to not implement fee-on-transfer behavior.

Administrative functions to add/remove privileged roles (`rely()` / `deny()`) exist.

### **SBEBeam**

Introduced in **Version 6** of this codebase, the SBEBeam contract allows permissioned facilitators to update the surplus processing parameters of the Kicker and the Splitter within bounds configured by the governance, without going through the full governance process for each adjustment.

Upon deployment, the immutable addresses of the Kicker and the Splitter (queried from the Kicker) are initialized in the constructor, and `minHop` is initialized to 5 minutes.



The contract implements two privilege levels: Wards (managed with `rely()` / `deny()`) administrate the contract and configure the bounds, while facilitators, called buds (managed with `kiss()` / `diss()`), may call `set()`.

The SBEBeam contract features the following state variables modifiable by the privileged wards:

- `maxKbump` - Maximum allowed value [rad] for `Kicker.kbump`.
- `minHop` - Minimum allowed value [seconds] for `Splitter.hop`. Cannot be set below 5 minutes.
- `maxRate` - Maximum allowed burn rate [rad/second], measured as `kbump / hop`.
- `tau` - Cooldown period [seconds] enforced between consecutive `set()` calls.
- `toc` - Timestamp of the last `set()` call.

A facilitator may call `set(kbump, burn, hop)`, which updates `kbump` on the Kicker as well as `burn` and `hop` on the Splitter in a single transaction and records the current timestamp in `toc`. Calling `set()` with unchanged values is valid and resets the cooldown. The call reverts unless all of the following conditions hold:

- The Splitter is live and not halted (`Splitter.hop < type(uint256).max`).
- The cooldown period has passed (`block.timestamp >= tau + toc`).
- `kbump` is at most `maxKbump` and is a multiple of RAY.
- `burn` is at most WAD (100%).
- `hop` is at least `minHop` and at most `5 * 365 days`.
- The burn rate `kbump / hop` is at most `maxRate`.
- Unless `burn == WAD`: A farm is configured in the Splitter.

If `burn < WAD` and `hop` differs from the current `rewardsDuration` of the farm, the `rewardsDuration` of the farm is updated to `hop` through the farm's owner. The owner is resolved dynamically by querying the farm configured in the Splitter, and is required to be a `FarmOwner` whose farm matches that same farm. For `set()` to succeed, the SBEBeam must be a ward of the Kicker, the Splitter, and, whenever the `FarmOwner` is invoked, the `FarmOwner`.

The upper bound on `hop` keeps it far below `type(uint256).max`, the value used to halt the Splitter, and limits the risk that re-rating the farm truncates its reward rate to zero. The flap threshold `Kicker.khump` is not modifiable through the SBEBeam.

## 2.3 Deployment Scripts

### 2.3.1 *FlapperDeploy*

Library `FlapperDeploy` implements the logic to be used off-chain to initiate the transactions that create the new contracts and switches its owner if necessary.

- `deployFlapperUniV2()` - deploys the Flapper (`FlapperUniV2SwapOnly` or `FlapperUniV2`) depending on the input flag `swapOnly` and transfers its ownership from the deployer to a designated owner.
- `deployOracleWrapper()` - deploys an `OracleWrapper`.
- `deploySplitter()` - deploys a Splitter and a `SplitterMom`. The ownership of both will be transferred from the deployer to a designated owner.
- `deployKicker()` - deploys the Kicker using the Vow and Splitter contract addresses retrieved from the chainlog as constructor parameters and transfers its ownership from the deployer to a designated owner.

- `deployFarmOwner()` - deploys a `FarmOwner` for the farm currently configured in the `Splitter` (retrieved from the chainlog) and transfers its ownership from the deployer to a designated owner.
- `deploySBEBeam()` - deploys the `SBEBeam` using the `Kicker` address retrieved from the chainlog as constructor parameter and transfers its ownership from the deployer to a designated owner.

## 2.3.2 *SplitterInstance*

A struct `SplitterInstance` is provided to wrap the `Splitter` and `Mom` addresses together after deployment.

## 2.3.3 *FlapperInit*

Library `FlapperInit` provides the logic to initialize the `Flapper`, `Splitter`, and `OracleWrapper` by configuring the parameters and access controls. This is expected to be used in a governance spell that is executed as `Delegatecall` by the `DSPauseProxy`.

**`initFlapperUniV2()`** will first conduct some sanity checks on the deployed `Flapper` to make sure it is deployed with correct parameters:

- `USDS`, `Spotter`, `pair`, `receiver` are set correctly (aligned with the passed `DSS` system and initialization configuration).
- `pair` is further checked that one of the tokens is `USDS` and the other one is the same as `Flapper.gem()`.
- `cfg.want` is ensured to be equal or above 90%.
- Then it sets `want` and `pip` of `Flapper`, grants `Splitter` the auth privilege over `Flapper`, and connects `Flapper` with `Splitter`.
- Finally it updates the address of `Flapper` in the chainlog in case `cfg.prevChainlogKey` is not 0.

In case a direct oracle is reused, `initDirectOracle()` will be called to grant `Flapper` the reading access to `pip`. Otherwise, `initOracleWrapper()` will be used to grant `Flapper` the reading access to the oracle wrapper and save the wrapper into the chainlog.

**`initSplitter()`** will conduct the following sanity checks on the deployed `Splitter` and `Mom`:

- Contracts `Vat` and `USDSJoin` are set correctly (aligned with the passed `DSS` system and initialization configuration). Ensures the `Splitter` is set correctly in `SplitterMom`.
- Some parameters (`hump`, `hop`, `burn`) of the input configuration `cfg` are further checked.
- Configures the `Splitter` by setting the `hop`, `burn`, and granting auth privilege to `Mom` and `Vow`.
- The `Vow` is configured with the new `Flapper`, `hump`, and `bump`.
- Sets the authority address of `Mom` to `MCD_ADM`.
- Eventually, it updates the chainlog with the new contracts.

**`setFarm()`** will do the following sanity checks and actions:

- Given a farm, its reward token is validated against the expected `USDSJoin`'s underlying token (`usds()`).
- Some sanity checks to ensure the `Splitter` has been initialized. Validates the `hop` matches the `Splitter`'s.
- Sets the farm on the `Splitter` with `file()`.
- Configures the `StakingReward` contract by setting its `setRewardsDistribution` to `Splitter` and reward duration to `hop`.
- Eventually, it updates the chainlog with the new farm.

**initKicker()** will first conduct some sanity checks on the deployed Kicker to make sure it is deployed with correct parameters:

- Validates that `cfg.kbump` is a multiple of RAY.
- Contracts `Vow` and `Splitter` are validated. `Vow` against the passed DSS instance and `Splitter` against the chainlog.

Then it will:

- Disable the `Vow`'s flop/flap auctions by setting `bump` (surplus auction lot size) and `dump` (debt auction lot size) to zero, and `hump` (surplus buffer) and `sump` (debt auction bid size) to `type(uint256).max`.
- Configure the Kicker by setting `khump` and `kbump` parameters.
- Grant `auth` privilege to Kicker on both `Vat` and `Splitter`.
- Finally, update the chainlog with the Kicker address using `cfg.chainlogKey`.

**initFarmOwner()** will:

- Validate that the `farm` of the `FarmOwner` matches the `farm` currently configured in the `Splitter` (retrieved from the chainlog).
- Transfer the ownership of the `farm` to the `FarmOwner` by calling `nominateNewOwner()` on the `farm`, followed by `acceptOwnership()` through the `FarmOwner`.
- Update the chainlog with the `FarmOwner` address using the given chainlog key.

**initSBEBeam()** will first conduct some sanity checks on the deployed SBEBeam:

- Contracts `Kicker` and `Splitter` are validated against the chainlog.
- The owner of the `farm` configured in the `Splitter` is validated to be the given `FarmOwner`.

Then it will:

- Configure the SBEBeam by setting `maxKbump`, `minHop`, `maxRate`, and `tau`.
- Grant the SBEBeam `auth` privilege on the `Kicker`, the `Splitter`, and the `FarmOwner`.
- Register the configured facilitators (`buds`) on the SBEBeam.
- Finally, update the chainlog with the SBEBeam address using `cfg.chainlogKey`.

Note that functions predating the SBEBeam (e.g. `setFarm()`) do not account for a live SBEBeam. The initialization functions are intended for the initial onboarding and are expected to be reviewed and potentially updated before any future use.

## 2.3.4 Changelog

**Version 2:** Cage functionality is removed from `Flapper`. Instead, a live flag is added and checked in `Splitter`.

**Version 3:** `Farm` can now be configured on `Splitter` instead of being an immutable.

**Version 4:** The rebranding of `MakerDAO` to `Sky` introduces `SKY` and `USDS`.

**Version 5:** The `Kicker` contract is introduced to replace the native `Vow.flap()` functionality.

**Version 6:** The SBEBeam contract is introduced, allowing permissioned facilitators to update the surplus processing parameters within bounds set by the governance. The `FarmOwner` contract is introduced to hold the ownership of the reward `farm`. Note that the `init` scripts have not been fully adjusted to incorporate the changes introduced in this version. If a new deployment is targeted, the current `init` scripts are expected to be adjusted or batched accordingly.

**Version 7:** The SBEBeam no longer stores a reference to the `FarmOwner`: The `farmOwner` state variable and its `file()` setter are removed, and `FlapperInit.initSBEBeam()` is adjusted

accordingly. Instead, `set()` resolves the owner of the farm configured in the Splitter dynamically: When `burn < WAD`, a farm must be configured in the Splitter, and before re-rating the `rewardsDuration`, the farm's owner is required to be a `FarmOwner` managing that same farm. Additionally, the constructor now emits a `File` event for the initial `minHop` value.

## 2.4 Trust Model

The surplus USDS available is split between the Flapper and the Farm. The Flapper is called "burn engine". All LP pool shares minted (`FlapperUniV2`) or Gem tokens (`FlapperUniV2SwapOnly`) are transferred to an external recipient set upon deployment. This recipient is intended to be a pause proxy controlled by the Governance. While the tokens are not actually burned, there is little difference between the protocol holding SKY compared to burning it. In doubt, the protocol is able to mint and sell SKY as it is to sell these tokens held at the pause proxy.

SKY Governance: fully trusted to set all parameters honestly and correctly. The file functions feature no sanity checks, the assumption is that the governance thoroughly checks the parameters before the transaction is executed. Will trigger the execution of the `FlapperInit` code executed as `delegatecall` in the context of the `GovernancePauseProxy`. Must ensure the parameters passed and the code to be executed is correct.

Callers of `Vow.flap()` or `Kicker.flap()` which invoke `Splitter.kick()` are untrusted.

Uniswap V2 is expected to work correctly as documented. The pool is expected to have sufficient liquidity, an illiquid pool may have negative consequences for this Flapper. Furthermore it is assumed the exchange rate on Uniswap v2 is synced with other markets.

Wards of `SplitterMom` and the price feeds are fully trusted.

Deployer: Untrusted. Deploys the contract using the `FlapperDeploy` script. Could modify the script and/or state of the deployment contract before transferring ownership.

For the Kicker introduced in [Version 5](#), the governance is further trusted to set the triggering threshold in the Splitter carefully and ensures that there is always enough surplus secured.

For the `SBEBeam` introduced in [Version 6](#), the facilitators (buds) are partially trusted: They may update `kbump`, `burn`, and `hop` within the bounds configured by the governance. Only the throughput-increasing directions are bounded (`kbump` is capped, `hop` is floored, and the burn rate `kbump / hop` is capped): Facilitators can hence stall the surplus processing (e.g. by lowering `kbump` or raising `hop`) and freely shift the split between the Flapper and the farm by changing `burn`. Such changes can be reverted by the governance. Facilitators cannot halt the Splitter, as `set()` caps `hop` at  $5 * 365$  days, far below `type(uint256).max`, and they cannot revive a Splitter that has been stopped or caged, as `set()` requires the Splitter to be live and not halted. The governance is further trusted to configure the bounds carefully. In particular, `maxKbump` must be a safe lot size on its own: As `hop` only enforces a minimum time since the last kick, a kick of the full `kbump` can be triggered immediately after a long idle period, regardless of `hop`. Wards of the `SBEBeam` and the `FarmOwner` are fully trusted. Additionally, it is expected that the reward tokens in farms are standard ERC-20 tokens (e.g. no fees, revert on transfer failure).

It is assumed that `Vat.flop()/Vat.flap()` are disabled by setting the parameters `bump` (surplus auction lot size) and `dump` (debt auction lot size) to zero, while setting `hump` (surplus buffer) and `sump` (debt auction bid size) to `type(uint256).max`:

- This effectively inhibits `flop/flap` auctions except at these maximum values (which are unreachable) with a lot size of zero.
- Otherwise, there might be a race condition between triggering `flap` from `Vow` or `Kicker` if their configurations are different. Inconsistency may exist where `flap` will succeed on one contract but revert on the other.

### 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:

- **Security**: Related to vulnerabilities that could be exploited by malicious actors
- **Design**: Architectural shortcomings and design inefficiencies
- **Correctness**: Mismatches between specification and implementation

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

|                                    |   |
|------------------------------------|---|
| <b>Critical</b> -Severity Findings | 0 |
| <b>High</b> -Severity Findings     | 0 |
| <b>Medium</b> -Severity Findings   | 0 |
| <b>Low</b> -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.

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|
| <b>Critical</b> -Severity Findings                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | 0 |
| <b>High</b> -Severity Findings                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           | 0 |
| <b>Medium</b> -Severity Findings                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | 0 |
| <b>Low</b> -Severity Findings                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | 6 |
| <ul style="list-style-type: none"><li>• <a href="#">Dangling Approval</a> <b>Code Corrected</b></li><li>• <a href="#">Incorrect Specification</a> <b>Code Corrected</b></li><li>• <a href="#">Initialization Does Not Check Divisor</a> <b>Code Corrected</b></li><li>• <a href="#">Missing Check for Bump</a> <b>Code Corrected</b></li><li>• <a href="#">Missing Check of Reward Token on Farm Contract</a> <b>Code Corrected</b></li><li>• <a href="#">Splitter.cage() Does Not Lock the Splitter</a> <b>Code Corrected</b></li></ul> |   |
| Informational Findings                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | 2 |
| <ul style="list-style-type: none"><li>• <a href="#">Initial minHop Is Set Without Emitting a File Event</a> <b>Code Corrected</b></li><li>• <a href="#">Live Flag Is Not Checked</a> <b>Code Corrected</b></li></ul>                                                                                                                                                                                                                                                                                                                     |   |

### 6.1 Dangling Approval

**Security** **Low** **Version 1** **Code Corrected**

CS-MKSPFL-007

In the constructors of both `FlapperUniV2SwapOnly` and `FlapperUniV2`, an approval is granted to `daiJoin` for moving its `vat.dai` balance, facilitating exiting or joining of Dai tokens (`vat.hope(address(daiJoin))`).

In the most recent version of these contracts, the interaction with `daiJoin` has been handed over to the `Splitter` contract. Now `Flappers` handle received ERC-20 Dai tokens directly, the dangling approval is never used in the current `Flapper`'s logic.

#### Code corrected:

The unnecessary `hope()` calls have been removed. At the same time `vat` and `daiJoin` (immutables) have been removed from `flapper` as they are no longer needed.

### 6.2 Incorrect Specification

**Correctness** **Low** **Version 1** **Code Corrected**

CS-MKSPFL-006

Library `Babylonian.sqrt()` provides an efficient way to approximate the square root of an `uint256`. Nevertheless, the following specification is incorrect because the last if branch incorrectly checks `xx` with `0x8`.

```
// this block is equivalent to r = uint256(1) << (BitMath.mostSignificantBit(x) / 2);
```

For instance, consider `x==1515`, its hexadecimal representation is `0x5eb` which has 11 bits in total. Thus its 0-indexed most significant bit divided by 2 results in 5 and `r` should be 32 ( $2^5$ ), while the following code yields 16 ( $2^4$ ).

```
// this block is equivalent to r = uint256(1) << (BitMath.mostSignificantBit(x) / 2);
uint256 xx = x;
uint256 r = 1;
if (xx >= 0x100000000000000000000000000000000) {xx >= 128; r <= 64;}
if (xx >= 0x10000000000000000000000000000000) {xx >= 64; r <= 32;}
if (xx >= 0x10000000000000000000000000000000) {xx >= 32; r <= 16;}
if (xx >= 0x10000000000000000000000000000000) {xx >= 16; r <= 8;}
if (xx >= 0x10000000000000000000000000000000) {xx >= 8; r <= 4;}
if (xx >= 0x10000000000000000000000000000000) {xx >= 4; r <= 2;}
if (xx >= 0x8) {r <= 1;}
```

`r` is used as the starting value for the iterative algorithm. While it may not be `uint256(1) << (BitMath.mostSignificantBit(x) / 2)` as specified, it is a sufficient approximation to the true square root, hence the algorithm's convergence remains unaffected.

---

#### Code corrected:

The code has been updated to match the behavior described in the original comment. The condition of the last if branch has been set to `xx >= 0x4`.

## 6.3 Initialization Does Not Check Divisor

Security Low Version 1 Code Corrected

CS-MKSPFL-008

The divisor is used to scale the MKR price to NGT (the re-denominated governance token). This is expected to match the rate stored in the MKR and NGT converter `MkrNgt.sol`.

This divisor is set by the (untrusted) deployer of the contract. However, this value is not checked during the contract's (trusted) initialization.

---

#### Code corrected:

The initialization code now features a sanity check for the divisor.

## 6.4 Missing Check for Bump

Correctness Low Version 1 Code Corrected

CS-MKSPFL-004

`initSplitter()` does not check if `bump` is a multiple of `RAY`. In case it is not, there would be a dust `vat.dai` balance on `Splitter` accumulating over time, which cannot be used up in `kick()`.

---

#### Code corrected:

A check has been added to verify that bump is a multiple of RAY.

## 6.5 Missing Check of Reward Token on Farm Contract

**Correctness** **Low** **Version 1** **Code Corrected**

CS-MKSPFL-005

`initSplitter()` does not check if the reward token of the farm contract matches the Dai token. In case it is not, `notifyRewardAmount()` may revert due to insufficient reward token sent to the farm contract, and Dai cannot be claimed as reward.

---

#### Code corrected:

The code of the initialization script now ensures that the farm's rewards token is DAI.

## 6.6 `Splitter.cage()` Does Not Lock the Splitter

**Design** **Low** **Version 1** **Code Corrected**

CS-MKSPFL-009

The Splitter distributes funds to two contracts. `Splitter.cage()` cages one of these contracts, the Flapper, by calling `FlapLike(flapper).cage()`. This sets the Flapper's state variable `live` to 0 and subsequent calls to the `flapper.exec()` will revert since the contract is no longer live.

Meanwhile, the Splitter contract itself is not caged. `splitter.kick()` can be executed but reverts upon calling `flapper.exec()`.

However, in case all funds are directed to the farming engine (`burn==0`), the call to `flapper.exec()` will be bypassed, and `splitter.kick()` can still be executed successfully to distribute funds to the farm.

---

#### Code corrected:

The `live` flag has been moved from Flapper to the Splitter. In case `Splitter.cage()` is called, the flag will be reset to 0 and the funds flow to both the burning engine and the farming engine will be stopped.

## 6.7 Initial minHop Is Set Without Emitting a File Event

**Informational** **Version 6** **Code Corrected**

CS-MKSPFL-011

The `SBEBeam` constructor initializes `minHop` to 5 minutes directly, without emitting an event. Every later update through `file("minHop", data)` emits `File("minHop", data)`, so off-chain

consumers that reconstruct the configuration from `File` events alone will miss the initial `minHop` value until it is set again via `file`.

---

**Code corrected:**

In `Version 7`, the constructor emits `File("minHop", 5 minutes)` when initializing `minHop`.

## 6.8 Live Flag Is Not Checked

Informational

Version 1

Code Corrected

CS-MKSPFL-001

All state except the `live` flag of the `Flapper` is checked in the initialization code.

---

**Code corrected:**

A sanity check has been added for the `live` flag.

## 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

### 7.1 No Flapper Sanity Check When Burn Is Enabled

**Informational** **Version 6** **Acknowledged**

CS-MKSPFL-010

`SBEBeam.set()` requires `farmOwner != 0x0` when `burn < WAD`, but there is no equivalent check for the burn side. If `flapper == 0x0` while `burn > 0`, `Splitter.kick()` would revert. Since `flapper` is similarly a parameter set by governance, mirroring the `farmOwner` check could prevent facilitators from enabling burn without a flapper and temporarily DoS-ing `kick()`.

---

#### Acknowledged:

No sanity check on the flapper side has been added in **Version 7**. Enabling burn while no flapper is configured makes `Splitter.kick()` revert until the parameters are corrected, without putting funds at risk.

### 7.2 maxKbump Is Not Bounded to Int256 Max

**Informational** **Version 6** **Acknowledged**

CS-MKSPFL-012

`SBEBeam.file("maxKbump", data)` sets `maxKbump` without an upper bound. `maxKbump` caps `kbump` in `set()` (`kbump <= maxKbump`), and `kbump` is later cast to `int256` by `_toInt256` in `Kicker.flap()`, which reverts above `type(int256).max`. So if `maxKbump` is filed above `type(int256).max`, a facilitator can set a `kbump` in that range and every `Kicker.flap()` reverts until `kbump` is lowered again. Bounding `maxKbump` at file time (e.g. `data <= uint256(type(int256).max)`) could prevent this.

---

#### Acknowledged:

The code is unchanged in **Version 7**. The governance is trusted to file a sensible `maxKbump` below `type(int256).max`.

### 7.3 No Revert Reason When SplitterMom Stops Splitter

**Informational** **Version 1** **Acknowledged**

CS-MKSPFL-002

SplitterMom can inhibit Splitter in an emergency. It does so by setting the minimum time between two executions of `kick()` to `type.max(uint256)`. `kick()` will then revert due to the addition overflow:

```
require(block.timestamp >= zzz + hop, "Splitter/kicked-too-soon");
```

Except when `kick()` has never been executed before and `zzz` is still equal to 0, the overflowing addition will cause the execution to revert and the `require` statement will not emit the message "Splitter/kicked-too-soon".

---

#### Acknowledged:

Sky states:

This is deemed an acceptable trade-off as this solution allows not having to read an additional storage variable.

## 7.4 Redundant daiJoin Field in Config

**Informational** **Version 1** **Acknowledged**

CS-MKSPFL-003

In `FlapperInit.sol`, two structs (`FlapperUniV2Config` and `SplitterConfig`) are passed as the initialization configuration for `initFlapperUniV2` and `initSplitter` respectively. Both contain the `daiJoin` address. `daiJoin` is already present in another input struct `dss` (`DssInstance`) however. It is redundant in the configuration.

---

#### Acknowledged:

Sky states:

This parameter is necessary to allow initialisation using `nstJoin` in the future. `nstJoin` will ultimately be added to `DssInstance` but we might need to deploy the `Splitter` before that happens.

## 8 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.

### 8.1 Call to `Vow.flap()` Can Be Sandwiched

**Note** **Version 1**

Parameter `want` of the Flapper contracts provides slippage protection regarding the reference price feed. However, depending on the `bump` and `want` parameters, as well as the current status of the pool, a `kick()` operation may be vulnerable to being sandwiched for arbitrage purposes. Analysis of the parameters and pool status to prevent arbitrage is out of scope for this review.

### 8.2 Deployment Verification

**Note** **Version 1**

Since deployment of the contracts is not performed by the governance directly, special care has to be taken that all contracts have been deployed correctly. While some variables can be checked upon initialization through the `PauseProxy`, some things have to be checked beforehand.

We therefore assume that all mappings in the deployed contracts are checked for any unwanted entries (by verifying the bytecode of the contract and then looking at the emitted events). This is especially crucial for the `wards` mapping.

### 8.3 Kicker Considerations

**Note** **Version 5**

Kicker replaces the native `Vow.flap()` as the entrypoint for surplus distribution to the `Splitter`. The following properties need to be considered by governance and dependent contracts:

1. It allows surplus distribution even when there is no actual system surplus (according to the `Vow`'s `dai` and `sin`) by setting a negative `khump`. As a consequence, the "surplus" USDS will be minted from thin air via `Vat.suck()`.
2. Unlike `Vow.flap()`, `Kicker.flap()` increases `Vat.sin(vow)` by `kbump` with each execution via `Vat.suck()`. When actual surplus exists in the `Vow`, this debt can be manually healed via `Vow.heal()`, but this requires an explicit call and is not automatic. When there is no actual surplus (enabled by setting negative `khump`), the minted USDS is unbacked and the debt accumulates. Contracts dependent on `Vat.sin(vow)` should be aware that this value will continuously increase with each kick, regardless of whether actual surplus is being distributed or unbacked USDS is being minted.

### 8.4 Oracle Rounds Down the Price

**Note** **Version 1**

In case the `gem` token used is a re-denominated version of an existing token, the `OracleWrapper` will scale down the price of an existing oracle (`pip`) by a certain divisor. If the return value from

`pip.read()` is not a multiple of `divisor`, the rescaled price may be lower than the actual price (due to precision loss in the conversion).

## 8.5 Semantics of Flapper in the System

### Note Version 1

The semantics of Flapper has slightly changed due to the introduction of the Splitter.

1. The Vow will be connected with the Splitter instead of directly to the Flapper. After a successful deployment and initialization, the Flapper state variable on Vow will return the Splitter address.
2. The interface of Flapper has been changed compared to the [docs](#). For instance, the surplus auction has been changed to swaps/deposits on UniswapV2. And the entry point has been changed from `kick()` to `exec()`.