PUBLIC

Code Assessment

of the Lockstake Smart Contracts

September 26, 2025

Produced for

Sky

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 12
4 Terminology 13
5 Open Findings 14
6 Resolved Findings 15
7 Informational 18
8 Notes 19

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 Lockstake according to Scope to support you in forming an opinion on their security risks.

Sky implements a staking framework that allows borrowing against governance tokens as collateral while retaining the ability to delegate their voting power and simultaneously allowing these tokens to be staked to earn yield.

The latest version reviewed version includes the new LockstakeCappedOsmWrapper, which reduces the risk of minting against price peaks linked to the low liquidity of the SKY token.

The most critical subjects covered in our audit are functional correctness, access control and integration with other contracts of the system. The general subjects covered are specification, complexity and unit testing. For the Lockstake implementation, Security regarding all the aforementioned subjects is high.

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	1
• Acknowledged	1

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 Lockstake 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	08 Apr 2024	a21be78009d80a94f1b84a44ecc04aef74fc40ea	Initial Version
2	16 Apr 2024	f07f45c0d1a47b9e3f82a1db29f4f800c21eaef3	After Intermediate Report
3	26 Apr 2024	ca0c577018dd664e2399e7d842364f2e5dac235f	Final Changes
4	05 July 2024	39dbea91e47911ddbf1122d5edd8c4a99934b059	Finalization
5	27 Aug 2024	7c71318623f5d6732457fd0c247a1f1760960011	Fix and Renaming
6	10 Apr 2025	ccc1c16b60a5eb30b4c5836ac93d63a138f70f54	SKY Migration
7	23 Apr 2025	9cb25125bceb488f39dc4ddd3b54c05217a260d1	Migrator Reset Line
8	02 Aug 2025	43662905a3504debc48d7ba3b3907c98fffb35f8	stUSDS Clipper
9	15 Aug 2025	d4dbe6eab1644e398d3fa59fe84c0522fad46532	stUSDS Clipper Fixes
10	22 Sep 2025	4df712d3a739a24718698f3ffdcf24f881769e98	LockstakeCappedOsmWrapper
11	26 Sep 2025	7db951621c7ac49e6d459a91ffbc0a8a43a4b12f	LockstakeCappedOsmWrapper Finalization

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`. Since Version 6 the `evm_version` is set to `shanghai`.

The following files are in scope of this review:

```
src/
    LockstakeClipper.sol
    LockstakeEngine.sol
    LockstakeMkr.sol
```

```
    LockstakeUrn.sol
    Multicall.sol
```

In Version 3, the deployment scripts have been added to the scope of the review:

```
deploy/
    LockstakeDeploy.sol
    LockstakeInit.sol
    LockstakeInstance.sol
```

Note that the contracts at Version 5 have been deployed.

The contracts in scope starting from Version 6 are:

```
src/
    LockstakeClipper.sol
    LockstakeEngine.sol
    LockstakeMigrator.sol
    LockstakeSky.sol
    LockstakeUrn.sol
    Multicall.sol
deploy/
    LockstakeDeploy.sol
    LockstakeInit.sol
    LockstakeInstance.sol
```

In Version 10, the following contract has been added to the scope of the review:

```
src/LockstakeCappedOsmWrapper.sol
```

2.1.1 Excluded from scope

All files not listed above including the tests are out of scope. The parameter selection is out of scope. The VoteDelegate and the farms have been part of other reviews. All other contracts are out of scope.

2.2 System Overview

This system overview describes the latest received version as well as the previously deployed version (Version 5) of the contracts as defined in the Assessment Overview.

At the end of this report section we have added a changelog for each of the changes accordingly to the versions. Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Sky implements a staking mechanism that allows participating in governance and yield farming while using the governance token as collateral for CDPs. Note that v1 is (being) deprecated and is using MKR while v2 is using SKY.

2.2.1 Lockstake v1: MKR

Note that this subsection only describes the Lockstake module up to Version 5 which corresponds to Lockstake v1 where MKR is used.

Sky introduces the **LockstakeEngine** that allows users to `open()` arbitrarily many **LockstakeUrn** contracts which will be used for the users' positions. Note that these urns can only be managed through the engine by the owner (urn deployer) or addresses authorized by the owner through the usual `hope()` / `nope()` mechanism.

To manage the urn's position, the following functionality is provided:

1. `lock()` and `lockSky()`: Users can deposit MKR as locked collateral (ink) to arbitrary urns of Lockstake ilk. The latter additionally converts SKY to MKR before locking the MKR.
2. `free()` and `freeSky()`: Similarly, urn-authorized addresses can unlock the collateral using these functions. Note that this collects an exit penalty in the form of burning parts of the MKR being unlocked.
3. `draw()`: Additionally, the urn-authorized addresses may `draw()` debt which mints the USDS token. This modifies the urn using `vat.frob()` which enforces the limits / minimum health factor.
4. `wipe()` and `wipeAll()`: Any address can repay debt partially or fully. Note that technically debt of addresses other than Lockstake urns can be repaid.

Locked MKR is typically held by the LockstakeEngine. Users can optionally move and thus delegate their locked MKR to one delegate (deployed by the vote delegate factory) per urn with `selectVoteDelegate()`. In this case, the staked MKR tokens will be transferred to the VoteDelegate contract, where they will be deposited and locked in the Chief. Locking and unlocking delegates to and undelegates automatically if a delegate is selected, the MKR tokens are moved accordingly.

Further, users' urns will hold **LockstakeMkr** tokens (ERC-20), a tokenized representation of an urn's MKR collateral locked. Note that it represents the `ink` and not the `gem`. As a consequence, during liquidations, the lsMKR will be burned on `kick()` since the LockstakeClipper will hold only a `gem` balance but not an `ink` balance. However, upon auction completion, lsMKR could be reminted due to adding the leftover collateral as `ink` to the urn.

Users can choose to deposit the lsMKR tokens into farms by selecting at most one farm per urn with `selectFarm()`. Note that farms have to be whitelisted by governance, managed with `addFarm()` and `delFarm()`. Locking and unlocking deposits and withdraws automatically (however deposits only work if the farm is still active). The farms will yield rewards that can be claimed anytime, even after unstaking or staking to another farm, through `getReward()`.

The LockstakeEngine supports `multicall()` to batch operations.

Liquidations of unhealthy urns will be initiated through `Dog.bark()` which invokes `Clipper.kick()` to start the Dutch auction. See our Liquidations 2.0 audit for a detailed description of how liquidations work. For the Lockstake ilk, a specialized **LockstakeClipper** which features callbacks to the engine is used. The LockstakeClipper works similarly and is different in the following aspects:

1. When an auction is started with `kick()`, the LockstakeEngine's `onKick()` hook is invoked to undelegate the MKR tokens, unstake and burn the LockstakeMkr tokens and track the auction (increases the urnAuctions counter, a non-zero urnAuctions counter disables delegating and staking).
2. The collateral transfer in `take()` is not implemented as a Vat-internal gem transfer with `Vat.flux()` but is rather implemented as a reduction of the LockstakeClipper's gem balance by the slice taken and calling the LockstakeEngine's `onTake()` hook which transfers out the collateral (MKR) directly.
3. The `ClipperCallee.clipperCall()` is disallowed on the LockstakeEngine.
4. When `take()` completes the auction (either all `tab` is covered or there is no more `lot` to sell) the LockstakeEngine's `onRemove()` hook is called. Parameters passed include the amount of MKR sold and left. `onRemove()` implements the functionality to calculate and burn the exit fee which applies, if sufficient funds are available, on liquidated urns (sold collateral / MKR) as well. Further, the urn's liquidation counter in the LockstakeEngine is reduced since the auction completed.

5. Upon `yank()` (cancelling an auction during `End.cage()`), `onRemove` reduces the urns auction counter.

Note that the LockstakeEngine further implements the standard authentication with `rely()` and `deny()`. The authorization allows setting the Jug with `file()`, add and deactivate farms (see above), and call the hooks. It is expected that the specialized clipper will hold the authorization. Additionally, there is a function `freeNoFee()` that allows an urn-authorized and authorized address to call free without collecting fees (e.g. in case of migrations). Further, LockstakeMkr implements the same authorization mechanism that allows to `mint()` (engine is expected to hold these rights).

2.2.1.1 Deployment

The Lockstake system is deployed in two steps:

1. Some EOA deploys the contracts and - if necessary - changes the owner of these contracts to the `PauseProxy`.
2. A governance `Spell` with quorum executes the initialization of the contracts through the `PauseProxy`.

LockstakeDeploy implements `deployLockstake()` to deploy a lockstake instance from an EOA using Foundry. A Lockstake instance consists of four contracts: LockstakeMkr, LockstakeEngine and LockstakeClipper and a calculator (for the Dutch-style auctions; deployed by the contract retrieved from the Chainlog with `CALC_FAB`) contract which are newly deployed.

The initialization is done by executing `initLockstake` of LockstakeInit in the context of the Governance Pause proxy. It implements the following steps:

1. The state of the given Lockstake instance is crosschecked with the LockstakeConfig passed as function argument `cfg`.
2. Sanity checks are performed on numeric parameters of the LockstakeConfig.
3. The new ilk is registered in the VAT, ward permissions are given to the LockstakeEngine and Clipper (Auction contract). Note that normally Clipper contracts do not get the ward role in the VAT, this special Clipper needs it due to different collateral management.
4. The `line` for the ilk is configured, the global `Line` is increased accordingly (`AutoLineLike.setIlk()`).
5. The rate module is initialized.
6. Spotter, Clipper, ClipperMom and End are added to the `bud` mapping of `PIP_MKR` (price oracle).
7. The OSMMom is given the ward role in `PIP_MKR`.
8. The spotter is configured (`mat`, `pip`) and updated (`poke()`).
9. Liquidation contract Dog is configured (`clip`, `chop`, `hole`), the LockstakeClipper is given the ward role.
10. LockstakeEngine is authorized on lsMKR.
11. The rate module `jug` is registered in the engine, farms are added. The LockstakeClipper is given the ward role in the engine.
12. LockstakeClipper is initialized (`buf`, `tail`, `cusp`, `chip`, `tip`, `stopped`, `vow`, `calc`). `upchost()` is triggered to update the cached `dust*chop` value. The Dog, End and ClipperMom are given the ward role in LockstakeClipper.
13. If provided, LineMom and ClipperMom are initialized.

2.2.2 Lockstake v2: SKY

Note that this subsection only describes the Lockstake module starting from Version 6 which corresponds to Lockstake V2 where SKY is used.

Lockstake for SKY is mostly equivalent to its predecessor. Besides renaming from MKR to SKY the most notable changes include:

- Fees are now immutable.
- `lockSky()` and `freeSky()` have been removed, given that MKR is not used in the new version of Lockstake. Similarly, the converter contract MkrSky have been removed from the contract.
- VoteDelegate v3 is used instead of the v2.

Since Version 8, Lockstake v2 works with stUSDS, where USDS is staked as segregated risk capital for borrowing backed by staked SKY in Lockstake. The new LockstakeClipper tracks debt under auction and can slash stUSDS deposits via `cut()` when bad debt occurs during liquidation.

2.2.2.1 Capped OSM Wrapper

In Version 10 a capped OSM wrapper was added for the SKY token. It wraps the current OSM while enforcing a cap. The goal is to limit minting against short term price peaks given the low liquidity of SKY. This mitigates minting risks but introduces potential risks for liquidations which are known and documented.

Generally, the same interface as an OSM is implemented with readers having to be authorized buds. Since it's a wrapper, it does not implement the usual privileged configuration functions of an OSM. Note that it is only intended to be used with `PIP_SKY` and recommended with liquidations off or in the contrary carefully evaluated the risk parameters. Liquidations for the Lockstake ilk are currently halted (the circuit breaker of the Lockstake clipper is currently set to 3 meaning liquidations are halted). In case of underwater positions exist it will be evaluated which process could be taken, e.g. offchain liquidations.

The initialization script replaces the current OSM for the Lockstake ilk in the spotter and ilk registry and sets the initial cap. It removes the spotter, clipper, clipper mom, and end from the buds list of the old OSM, and adds them to the buds list of the new capped OSM. It adds the capped OSM to the buds of the old OSM allowing the wrapper to read the price.

2.2.2.2 Migration

A new contract, **LockstakeMigrator**, facilitating the migrations of urns from the v1 to v2 has been introduced. The contract exposes the function `migrate()` to users which can operate as follows:

- If an urn has no debt, collateral is freed (without fee) from the old engine so that MKR can be converted to SKY and locked in the new one.
- If an urn has debt, a `Vat.dai` flashloan is taken. First all debt is wiped. Second the collateral is migrated as described above. Last, the flashloan debt is settled by drawing debt accordingly using the new engine.

The new ilk is assumed to have 0 line configured in the vat, and the migrator will lift the line to 55M temporarily in each migration.

Note that only addresses authorized for both source and target urn are allowed perform operations with the urns.

2.2.2.3 Deployment

The deployment script additionally deploys LockstakeMigrator besides LockstakeSky, LockstakeEngine, LockstakeClipper and the calculator. Otherwise, the deployment is equivalent to the deployment of v1 (however, adjusted for new code).

The initialization of the new version is mostly equivalent to the prior version. The most notable differences beside renaming of variables are:

- The oracle used in steps 6 and 7 (of the v1 deployment) is not `PIP_MKR` but `PIP_SKY`.
- The v1 addresses are migrated to new identifiers (`<previous identifiers> | _OLD_V1`) while v2 uses the same identifiers as the previous version used. The only exception is for the LockstakeSky token which is stored with `LOCKSTAKE_SKY` (`LOCKSTAKE_MKR` does not exist anymore in the Chainlog).
- The migrator is sanity checked and is granted authorization on the v1 engine to be able to call `freeNoFee()`.
- The migrator is also granted ward role on the vat for temporarily lifting the new ilk's line.
- The initialization script disables borrowing for the v1 ilk by setting the `Vat.line` for the ilk to zero.

In **Version 8**, `deployClipper()` is added to the deployment library to deploy a new LockstakeClipper that tracks auction debt and manages stUSDS slashing.

It is initialized with `updateClipper()`. Assuming the liquidation is paused and there is no ongoing liquidation, the existing Clipper for lssky will be detached, and the new Clipper will be attached, namely:

- Parameters are configured and contracts are wired with each other.
- Necessary roles are granted between the Clipper and other system components such as Dog, Vat, Osm, and End.
- Updates are made in ilkRegistry and Chainlog to reflect the changes.

Note liquidation is stopped (level==3) on the new clipper. Function `enableLiquidations()` is provided to enable the liquidations later and enable ClipperMom on the Clipper.

2.2.3 Changelog

Since **Version 2**, the locking functions (collateral top-ups) are not permissioned anymore but only ensure that the urn address is a LockstakeUrn.

In **Version 3**, deployment scripts have been added to the scope.

In **Version 5**, NST and NGT has been renamed to USDS and SKY respectively. In addition, the following changes are made:

1. `CREATE2` has been replaced to `CREATE` when creating a new `urn`.
2. As a consequence, an additional mapping `ownerUrns` is provided to retrieve an `urn` address by an `owner` and an `index`. Hence, function `getUrn()` is removed.
3. `fee` is no longer immutable and is by an auth-ed function `file(bytes32 what, uint256 data)`.
4. Instead of specifying the `urn` as a parameter for the functions, the `owner` and `index` are specified.
5. Function `selectVoteDelegate()` will call `jug.drip(ilk)` to use the up-to-date rate when checking if the urn is safe.
6. Under certain conditions, `vat.frob()` in function `onRemove()` reverted. To prevent reverts on dusty urns, `vat.grab()` is now used instead.
7. The deployment and initialization script have been adjusted accordingly. Note since the `fee` is no longer set in the constructor (default to 0), it is now configured in the initialization script.

In **Version 6**, a new version based on SKY has been introduced. More details can be found in the respective section.

In **Version 7**, the line will not be configured for the new ilk during initialization, and the migrator will lift the line to 55M temporarily in each migration for the existing positions in Lockstake v1.

In Version 8, the LockstakeClipper contract is updated to enable stUSDS deposit slashing. The existing Clipper will be detached and replaced with the new one.

In Version 10, the LockstakeCappedOsmWrapper will replace the current OSM for the SKY token. It returns the minimum of the price reported by the current OSM and the configured `cap`.

2.3 Roles and Trust Model

The following roles are defined:

1. Governance: Authorizer of addresses. Fully trusted. Could mint unbacked LockstakeMkr/LockstakeSky tokens or call the hooks arbitrarily leading to unexpected behaviour. For example, MKR/SKY could be transferred out of the system.
2. Farms: Trusted. Expected to be the Endgame Synthetix-like staking contracts. We expect that the farms correspond to the contracts reviewed in the Endgame Toolkit Audit. Could move and reuse the lsMKR/lsSky tokens if malicious.
3. Delegates: For v2 of Lockstake, we expect VoteDelegate v3 (referencing the new Chief contract). For v1 of Lockstake, we expect VoteDelegate v2 that implements an on-demand window for freeing due to flashloan protection on DsChief potentially delaying liquidations. Please see our VoteDelegate Report for more details.
4. Other authorized addresses: Trusted to implement the proper logic.
5. Users: Untrusted.

Additionally, we expect that the setup is performed accordingly, see note Setup for some more details.

Since Version 8, it is further assumed the LockstakeMigrator is deprecated (as August 2025 it is no longer a ward on Vat). Otherwise, it could directly change `ilk.line` during migration.

For the `LockstakeCappedOsmWrapper` (introduced in `|ver10|`), it is assumed it is only used for `PIP_SKY` and the `cap` is set properly at all times. Note that use is recommended with liquidations off. Otherwise the risk parameters must be carefully evaluated. Urn owners should be aware of the potential liquidation loss since the SKY collateral might be under-valued if the cap caps the real price of the SKY token. In the worst case, the governance can set a low `cap` when liquidation is on, hence liquidating all active SKY positions. The capped SKY price also applies in the End process, which may prevent fixing an excessively high SKY price peak but could result in a too low price if the cap is below the true SKY value. Should the end process be considered, this needs to be evaluated carefully.

Deployers are supposed to deploy the contracts as specified by the reviewed script. Deployers are, however, EOAs that could perform undesired actions besides simple deployment, such as changing the settings of the system or granting themselves special privileges. It is important that after deployment, concerned parties thoroughly check the state of the deployed contracts to ensure that no unexpected action has been taken on them during deployment.

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	1

- Rate and urnImplementation Not Validated Acknowledged

5.1 Rate and urnImplementation Not Validated

Correctness **Low** **Version 3** **Acknowledged**

CS-MLS-001

During initialization, all state is validated for consistency. While voters must ensure the deployment is legitimate, the expected bytecode has been deployed and the expected constructor code has been executed, the initialization code should validate all state variables to be set correctly.

`LockstakeInit.initLockstake()` does not validate the correct setting of the `rate` parameter nor the `urnImplementation`.

Acknowledged:

Sky states:

```
The goal of the deployment scripts validations is to only check the constructor params.
Anything else that is done in the deployment or afterwards (including internal immutables
setting and storage writes) are out of scope and are assumed to be checked separately.
Some init scripts might validate more, but that is considered nice-to-have at best.
```

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	1

- Line Is Temporarily Under-Estimated upon Slashing Code Corrected

Informational Findings	5

- Clip Update in Ilk Registry Code Corrected
- Inconsistent line Update Specification Changed
- Outdated Comment Specification Changed
- Specification Mismatches Specification Changed
- Unpermissioned Collateral Top up Code Corrected

6.1 Line Is Temporarily Under-Estimated upon Slashing

Correctness Low Version 8 Code Corrected

CS-MLS-005

In Lockstake Clipper, if bad debt occurs during a liquidation (`take()`), deposit on stUSDS will be slashed with `cut()` to cover it, which also updates the `line` of the ilk:

```
// In stUSDS.sol
vat.file(ilk, "line", _min(line, _subcap(totalSupply * chi, clip.Due())));
```

However, this update in `cut()` uses the old `clip.Due()` before `Due` is decreased by the total debt wiped in this `take()`:

```
if (due > owe && cuttee != address(0)) {
    CutteeLike(cuttee).cut(due - owe);
}

Due -= due;
```

Consequently, the `line` will be under-estimated. Though a consecutive user operation on stUSDS (i.e. `drip()`, `deposit()`...) will correct the `line`.

Code corrected:

Code has been corrected to reduce the `Due` before calling `cut()`, hence updating the `line` correctly.

6.2 Clip Update in Ilk Registry

Informational | Version 8 | Code Corrected

CS-MLS-006

In the initialization library, `updateClipper()` calls `ilkRegistry.put()` to update the clip address of the ilk. Since only the clip is being changed, `ilkRegistry.file("xlip")` may be used instead of overwriting all the ilk configs.

Code corrected:

Instead of overwriting all configs, it now uses `file(ilk, "xlip", address(se.clipper))` to only change the clipper.

6.3 Inconsistent `line` Update

Informational | Version 8 | Specification Changed

CS-MLS-007

In Lockstake Clipper, when a new liquidation is trigger (`kick()`), `drip()` will be called on cuttee (stUSDS) to sync the `line`. In the scenarios outlined below, the `line` may also need to be synced since `Due` is modified:

1. In `take()` when there is no stUSDS slashing.
2. In `yank()`.

This can be corrected by anyone with a following call to `drip()` (or any other user operation that updates `line`).

Specification Changed:

Additional comments have been added in `take()` and `yank()` clarifying the requirement to call `cuttee.drip()`.

6.4 Outdated Comment

Informational | Version 8 | Specification Changed

CS-MLS-008

The comment in `LockstakeClipper.take()`

```
// Do external call (if data is defined) but to be
// extremely careful we don't allow to do it to the three
// contracts which the LockstakeClipper needs to be authorized
DogLike dog_ = dog;
if (
    data.length > 0 &&
    who != address(vat) &&
```

```
    who != address(dog_) &&
    who != address(engine) &&
    (who != cuttee || cuttee == address(0)) // Keep consistency executing with address(0) to revert
)
```

has not been updated to include the new `cuttee` and still reads three contracts instead of four.

Specification changed:

The comment has been corrected.

6.5 Specification Mismatches

Informational **Version 1** **Specification Changed**

CS-MLS-003

The README specifies the module. Below is a list of mismatches:

1. `selectDelegate` is defined in the user-facing functions of the LockstakeEngine. However, `selectVoteDelegate` is the function name.
2. `wipeAll` is undocumented.
3. `LockstakeClipper.stopped` is a configurable parameter that is undocumented while `LockstakeClipper.chost` is updated by function `upchost` and not by `file`.

Specification changed:

The README has been updated accordingly.

6.6 Unpermissioned Collateral Top up

Informational **Version 1** **Code Corrected**

CS-MLS-004

Unlike direct interactions with `VAT.frob()`, the LockstakeEngine does not permit unauthorized increases in (locked) collateral.

Code corrected:

The locking functions (collateral top-ups) are no longer permissioned. The internal `_lock()` now ensures that the urn address is a LockstakeUrn.

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 Default Cap of 0

Informational **Version 10**

CS-MLS-009

In the constructor of the Capped Osm Wrapper, no cap is set, hence the initial cap is 0. This is not a safe default since the OSM wrapper returns min(cap, price). Without a configured cap, it would return a price of zero. Note that a price of 0 inhibits drawing more debt or starting auctions in the clipper.

The provided initialization script explicitly sets a cap in `updateOsm()` during initialization, so this is not an issue in this project. For future reference, users of this code should be aware that the default unconfigured value means a price of 0.

7.2 Locking Discrepancy With Zero

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

CS-MLS-002

The semantics of locking and unlocking an amount of zero are different when an urn has a farm and when it does not. Namely, the operations will revert when a farm is selected and will not when no farm is selected due to the farms reverting when zero amounts are staked/unstaked.

Acknowledged:

Sky replied:

```
Considering the asymmetry is happening due to some StakingRewards code, we are fine leaving it as it is.
```

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 Capped Osm Wrapper Interfaces

Note Version 10

The Capped OSM Wrapper is a wrapper that relays OSM prices and can be used as an OSM in the system. However, as a wrapper it does not implement the following standard OSM interfaces:

1. Privileged functions do not exist: `stop()`, `start()`, `change()`, `step()`, and `void()`.
2. Batched `kiss()` and `diss()` functions are not available.

8.2 Considerations for Migrations

Note Version 6

Users: Prior to using the Lockstake Migrator to migrate from v1 to v2, an user has to perform a set of action:

- `hope(migrator)` must be called in the old and new engine, respectively, for the source and the destination urn (note that for the destination urn this might be overly restrictive sometimes as outlined in the README).
- The destination urn must be created with `open`.

Further, note that if an urn is being liquidated, migrating the urn might result in needing to migrate again.

Users should be aware that migration might fail under certain circumstances. However, the migration can be performed manually.

In addition, the migration does not include the farm and vote delegate selection, hence users need to select them manually after the migration.

Governance: Governance should be aware that if `MkrSky` fees are enabled, the migrator will fail to perform the operations. Hence, when activating fees in that contract, the migration of Lockstake should be considered. Note that other such configurations exist (e.g. `line` set to zero).

Further, governance should be aware that `VOTE_DELEGATE_FACTORY` and `MKR_SKY` should be updated in the Chainlog prior to the LockstakeInit library being used. Additionally, it is expected that `PIP_SKY` will be configured prior to the execution of the library's code.

In addition, the fee and configuration of the new ilk (SKY) may influence the migration incentives if they are different to the existing ones.

8.3 Deployment Verification

Note Version 3

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 the initcode, bytecode, traces and storage (e.g. mappings) are checked for unintended entries, calls or similar. This is especially crucial for any value stored in a mapping array or similar (e.g. could break access control, could lead to stealing of funds). Additionally, it is of utmost importance that no allowance is given to unexpected addresses (e.g. MKR/SKY approval to arbitrary addresses could have been given in the constructor).

8.4 End Considerations

Note Version 1

The LockstakeClipper implements `yank()` which is required by the shutdown process. Note that the emergency shutdown is unsupported (as documented) since the `gem` balance that would be received during shutdown is not redeemable and thus the MKR tokens would not be able to be exited from the LockstakeEngine. A governance-assisted shutdown, which must be carefully planned, can be possible. For this, a mechanism must be implemented in order to make the collateral redeemable.

Further, `yank()` will not collect any fees on collateral already sold.

8.5 Governance Token as Collateral

Note Version 1

The Lockstake contracts allow for borrowing USDS (so-called NST in before Version 5) against the governance token. Governance should carefully set the ilk parameters (e.g. debt ceiling) due to the potential correlation between the price of the governance token and the price of the USDS. In tumultuous situations, when the USDS loses its peg, the price of the governance token could drop as a consequence (e.g. if liquidations fail to complete in time or successfully). As a consequence, the price of the USDS could further depeg leading to a vicious circle. Ultimately, governance should ensure that parameters and collateral diversity limit the risk.

It's worth mentioning since Version 8 USDS deposited in stUSDS is expected to fund the staked SKY backed borrowing, hence mitigating the risks of incurring system bad debt. For more details please refer to the stUSDS Review.

8.6 Ink Token During Liquidations

Note Version 1

lsMkr/lsSky is minted to an urn when `ink` is added to a LockstakeUrn position and burned when `ink` is removed from such a position. Ultimately, it is a tokenization of `ink` of LockstakeUrn. When the collateral is seized during liquidation and moved to the Clipper, the lsMkr/lsSky is burned. Should there be leftover collateral after the auction concludes, the ink is moved back to the LockstakeUrn and the respective amount of lsMkr/lsSky is minted again.

8.7 Migration Line Should Not Be Exceeded

Note Version 7

In this version, the migrator will lift the line of new ilk to 55M temporarily in each migration of the existing positions in Lockstake v1. With this design, it should be ensured that the line of ilk `LSE-MKR-A` should be below 55M with sufficient room preserved for i.e. stability fees. By the time of this review (April 2025), the `VAT.line`, `AUTOLINE.line`, and `AUTOLINE.gap` of `LSE-MKR-A` are all 45M.

8.8 One Hour Update Delay Is Not Guaranteed

Note Version 10

OSM enforces a one-hour delay before the next value becomes effective. However, due to the Capped OSM Wrapper design, the price returned from `read()` may change within 1 hour if a new cap is set. Further, the price returned from `read()` may not change after one hour if both the current and the next price exceed the cap.

8.9 Setup

Note Version 1

The deployment is expected to be trusted and the parameters are expected to be set accordingly.

The following authorization should be given by the governance.

1. LockstakeClipper should be authorized on LockstakeEngine.
2. LockstakeEngine should be authorized on lsMKR/lsSky contract to be able to mint.
3. Dog should be authorized on LockstakeClipper.

Additionally, it is expected that the liquidation parameters are set so that exit fees can be taken on liquidations, too. Note that `README.md` outlines parts of this. If that is not the case it could be profitable to self-liquidate to bypass the exit fees which would violate the specification.

8.10 Token Received in Liquidation Auction

Note Version 1

Liquidators should be aware that for this special ilk, they will not receive a `gem` balance in the VAT but will receive MKR/SKY tokens directly when buying collateral in auctions.