PUBLIC

Code Assessment

of the Lockstake Smart Contracts

July 30, 2024

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help MakerDAO 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.

MakerDAO 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 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.

Before the Governance initializes the Lockstake instance the deployed contracts must be validated carefully. Please refer to note Deployment verification for more details.

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	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 April 2024	a21be78009d80a94f1b84a44ecc04aef74fc40ea	Initial Version
2	16 April 2024	f07f45c0d1a47b9e3f82a1db29f4f800c21eaef3	After Intermediate Report
3	26 April 2024	ca0c577018dd664e2399e7d842364f2e5dac235f	Final Changes
4	05 July 2024	39dbea91e47911ddbf1122d5edd8c4a99934b059	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`.

The following files are in scope of this review:

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

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

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

2.1.1 Excluded from scope

All files not listed above including the tests are out of scope. Before Version 3 the deployment scripts have been out of scope. The parameter selection is out of scope. VoteDelegatee has been covered as part of another review. The Farms and the main DAI Stablecoin System are not part of this review.

2.2 System Overview

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

MakerDAO implements a staking mechanism that allows participating in governance and yield farming while using MKR as collateral for CDPs.

More specifically, MakerDAO 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 `lockNgt()`: Urn-authorized addresses can deposit MKR as locked collateral (ink) for their specified urn of Lockstake ilk. The latter additionally converts NGT to MKR before locking the MKR.
2. `free()` and `freeNgt()`: 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 NST 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 VoteDelegatee 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 to set 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 Changes in Version 2

The locking functions (collateral top-ups) are not permissioned anymore but only ensure that the urn address is a LockstakeUrn.

2.2.2 Changes in Version 3

Deployment scripts have been added to the scope of the review.

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.
14. The new ilk and addresses of the Lockstake instance are set in the chainlog.

2.2.3 Roles and Trust Model

The following roles are defined:

1. Governance: Authorizer of addresses. Fully trusted. Could mint unbacked LockstakeMkr tokens or call the hooks arbitrarily leading to unexpected behaviour. For example, MKR 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 tokens if malicious.
3. Delegates: We expect the vote delegate contract to be a version that implements a mechanism as described in the README in section 3a to protect from the outlined delayed liquidation attacks.
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.

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 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:

MakerDAO 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 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	0
Informational Findings	2

- Specification Mismatches Specification Changed
- Unpermissioned Collateral Top up Code Corrected

6.1 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.

6.2 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 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:

MakerDAO 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 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 approval to arbitrary addresses could have been given in the constructor).

8.2 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 implmented in order to make the collateral redeemable.

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

8.3 Governance Token as Collateral

Note Version 1

The Lockstake contracts allow for borrowing NSTs 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 NST. In tumultuous situations, when the NST 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 NST could further depeg leading to a vicious circle. Ultimately, governance should ensure that parameters and collateral diversity limit the risk.

8.4 MKR 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 tokens directly when buying collateral in auctions.

8.5 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 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.

8.6 lsMKR During Liquidations

Note Version 1

lsMKR 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 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 is minted again.