PUBLIC

Code Assessment

of the DSS Crop Join Smart Contracts

March 1, 2022

Produced for

MAKER

by

CHAINSECURITY

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 DSS Crop Join according to Scope to support you in forming an opinion on their security risks.

DSS-Crop-join introduces support for new ilks with a join adapter facilitating the staking of the collateral tokens in a third party system to generate reward instead of simply holding the tokens at the join adapter. The generated reward is distributed amongst the users the collateral belongs to.

The most critical subjects covered in our audit are the security of the new contracts, the functional correctness and the impact of these changes on the existing system.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	6
• Code Corrected	4
• Specification Changed	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 DSS Crop Join 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	13 January 2022	8a2ececd8d92a833757b0f725eb3640c9be6646c	Initial Version
2	25 January 2022	546fcb5463c6f543dda3bd2302b5a408e3892b15	After Intermediate Report
3	15 February 2022	beb74bfaa9f0392cf5e973a5165372dc87f239de	Proxy Actions Extension
4	28 February 2022	b2964768265de0c72633257b387d6d9d6e286de0	Further Changes

For the solidity smart contracts, the compiler version `0.6.12` was chosen which was the default compiler version used in the Maker ecosystem when these contracts were developed.

Following files where in scope of this review:

- SynthetixJoin.sol
- CropJoin.sol
- Cropper.sol

For the liquidation and the corresponding auction of the collateral a modified version of the Clipper contract is to be used.

- ProxyManagerClipper.sol - The diff to Clip.sol of liquidations 2.0 only. The changes in the liquidation process related to CropJoin has been reviewed.

The main code of Clip.sol is not part of this review but has been reviewed as part of the liquidations 2.0 review.

CropJoin has been reviewed under both viewpoints, first as a part of SynthethixJoin which inherits CropJoin and secondly, on a best effort basis as a base contract to be inherited by further adapters connecting to other reward systems.

In version 3, DssProxyActionsCropper.sol was added to the scope.

2.1.1 Excluded from scope

SushiJoinV1.sol and SushiJoinV2 where not part of this review.

3 System Overview

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

At the end of this report section we have added subsections 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.

DSS-Crop-join introduces support for new ilks with a join adapter facilitating the staking of the collateral tokens in a third party system to generate reward instead of simply holding the tokens at the join adapter. The generated reward is distributed amongst the users the collateral belongs to.

3.1 Contracts:

3.1.1 CropJoin

This contract serves as the basis for the individual join adapters tailored for the reward system to be supported and is meant to be inherited from. It contains the logic for managing the accounting of stakes and rewards for a given `ilk` across the urns. Its internal function `crop` must be overridden and implement the mechanism that will collect the reward from the reward contracts. The public function `nav` must be overridden.

Contrary to the traditional GemJoin adapters which users can directly interact with, in order to account for the proper distribution of the accrued rewards user can only interact with such an ilk using the Cropper contract and a urnproxy.

Functions responsible for the transfer of `gem` and the accounting, for a given `urn`:

- `join` Deposit of collateral. Will first call crop to accrue reward and update the accounting before processing the new contribution. The staked amount is accounted for as a rounded-down amount of shares.
- `exit` Withdrawal of free collateral. Will first call crop to accrue reward and update the accounting before processing the new contribution. The amount to be unstaked is accounted for as a rounded-up amount of shares.
- `flee` will remove the full stake of the user from the system represented by the `ilk` and update the accounting. If one flees, its bonus is lost forever. Intended to be used only in case of a problem with the rewards. Unclaimed rewards of this user accounted for in `stock` remains forever locked.

An additional function, `tack` is used to adjust the accounting of `stake` and `crops` after some collateral have been transferred in the VAT. The function does not perform any access control check but instead ensures that the final state of the VAT and the updated accounting in `CropJoin` match.

The `cage` function can be used by authorized callers to set the join adapter to `live = 0`.

3.1.2 SynthetixJoin

Join adapter for the Synthetix Staking Rewards contract, wrapping `CropJoin`. It extends `CropJoin` and has an additional `uncage` function that can reset `live = 1`. The Synthetix staking rewards mechanism is as well used by other such as UniswapV2 and LIDO which this join adapter can be used for.

Pausing (caging) the system withdraws all staked tokens. Exit and flee are still possible in this state. The SynthetixJoin adds an uncage functionality to reactive the adapter, this stakes all available tokens.

3.1.3 Cropper

Due to the accounting of the accrued rewards interaction with `gems` of this type of ilk must be done through a gateway contract. The gateway contract wraps all required functionalities (normal operation of the VAT, liquidations and shutdown) while handling the accounting of the rewards. Hence users cannot use this `ilk` on an urn they control but only through the Cropper contract which deploys an urnproxy for each user.

Users can also manage their urn proxy via the `Cropper` with:

- `move` will call the `move` function of the `vat` to transfer DAI balance in the internal accounting of the VAT from the urn proxy to an arbitrary address.
- `frob` will call the `frob` function of the `vat` to modify the urn proxy vault and ensure the accounting in CropJoin is updated. For the parameters u and v the urnproxy of u (which must be equal to v) is passed.
- `flux` will call the `flux` function of the `vat` to transfer collateral, this will trigger the `tack` for accounting on the `CropJoin`. Only calling `flux` will also transfer the uncollected reward to the destination urn. If the source urn wants to collect the reward before transferring its stake, user must call `join` with `val = 0` to harvest and then `flux`.

Note that VAT.fork() (splitting a vault) is not exposed to the user and cannot be reached outside of `quit()`.

Considerations for liquidations:

The liquidation 2.0 system (Dog) is used. Dog.bark() is used to liquidate a vault and initiate the start of the auction. For the auction of such an ilk a special auction (Clipper) contract called ProxyManagerClipper is used which ensures the accounting of the rewards is updated in line with the change of the ink/gem at the VAT.

Dog.bark() liquidates the urn and assigns the ink/art to the Clipper contract using `Vat.grab()` before starting the auction by calling `ProxyManagerClipper.kick()`. The ProxyManagerClipper's `kick` function contains an additional callback to `Cropper.onLiquidation()` to update the accounting: first the user's reward is collected and paid out. Next the accounting is updated, meaning the seized amount of collateral is assigned from the user to the Clipper contract.

When a bidder buys some of the collateral in the Dutch style auction using `ProxyManagerClipper.take()` the collateral is transferred to the urnproxy of the address passed as parameter who and the accounting is updated through a callback to `Cropper.onVatFlux()`.

Bidders on auctions of these ilks need to pass an address as parameter who which already owns an UrnProxy at the Cropper or the execution of `take()` will revert. This prevents unaware buyers to participate in such an auction and therefore receiving collateral on an urnproxy which can only be accessed via the ProxyManager (Cropper).

Leftover collateral not needed to cover the debt is returned to the liquidated urn and the accounting regarding the rewards is updated using a call to `Cropper.onVatFlux()`.

In summary, for the liquidation process, the Cropper contract exposes the following functionalities:

- `onLiquidation`: will collect and send the reward to the user for the vault being liquidated and transfer some the stake to the Clipper (msg.sender). By calling this function with `wad = 0`, it is possible to force push its rewards to any existing user.
- `onVatFlux`: updates the `CropJoin` accounting if collateral has been moved from an urn proxy to another in the `vat`

These two functions are publicly callable but will only succeed if the `ilk` has been transferred on the VAT accounting, which can only be done by authorized parties upon vault liquidation for those special urn proxies.

Considerations for VAT shutdown:

The restrictions of the ilks managed by the Cropper impact the shutdown process.

During the step `freeEth` / `feeGem()` only vaults holding this type of collateral are affected hence the user is aware of the special behavior. In order to successfully execute `end.free()` the urnproxy has to be temporarily forked into a normal urn. The functions in DssProxyActionsEndCropper do this.

In the final step of the shutdown process everyone can retrieve gems from all ilks. Users unaware of the specialties of these ilks may participate. Due to the managed gem join adapter, one cannot simply exit the collateral. Exiting the collateral can only be done through the Cropper and for this the gem must be in an urn proxy.

`cashGem/cashETH()` of the DssProxyActionsEndCropper implement the necessary steps. These functions can be used for vaults managed through a DsProxy. For vaults which are managed directly the respective steps must be done similarly.

`move`, `frob`, `flux` and `quit` allow delegated actions.

3.2 Trust Model & Roles

All implementation contracts are upgradeable.

Wards: Fully trusted to behave honestly and correctly at all times. They can update the implementation of the Cropper contract. They can update the implementation of the CropJoin contract and call `join`, `exit`, `flee`, `cage` and `uncage` of SynthetixJoin. Note that the Proxy and Implementation contracts share the same mapping for the wards.

Users must fully trust the Cropper and CropJoin contracts as they feature full access on the user's vault in the VAT. Due to the upgradeability this requires users fully trust the wards as they have the power to change the implementation code of the Cropper and CropJoin contracts.

Users are not trusted and this is mitigated by the use of urn proxies. The Maker' system contracts are fully trusted. The reward mechanism of the reward contracts is semi-trusted to work as intended, to mitigate the risk of locked tokens due to the reward contract the `flee` function can be called. The extern staking system is trusted to work as intended and allow staking and withdrawal at any point in time.

Tokens: The Cropper disregards the return value of the ERC20 `approve()` and `transferFrom()` calls. In the Maker system the adapter (join) will take care of the token idiosyncrasies and manage the collateral tokens. The CropJoin contract reviewed expects and handles the return values of the ERC-20 tokens. This doesn't work for all supported collateral token in the Maker system, USDT which is not compliant with the ERC-20 standard as it does not feature a return value on transfers will not work with this CropJoin contract. Generally, it is assumed that only trusted tokens are added to the Maker system. All the `gem` and `bonus` are assumed to be ERC20 tokens and are expected to work correctly, without special behaviors.

3.3 Changes in Version 3

The `flee()` functions now take an additional parameter `uint256 val` which allows to specify the amount to exit. This change was necessary to allow `cashETH() / cashGem()` of the DssProxyActionsEndCropper to pass the amount to exit and hence behave similarly as the original proxy actions contract.

3.3.1 DssProxyActionsCropper

In version 3, the contracts DssProxyActionsCropper was added to facilitate the interaction with the core functionality. This is an adapted version of the DssProxyActions contract.

The intended interactions is that a user calls his deployed DSProxy that delegatecalls to the DssProxyActionsCropper contract which interacts with the Cropper. Note that the DSProxy contract of the user is the owner of the UrnProxy in the Cropper.

Following functionality are implemented:

- `lockETH()`: Allows users to send Ether, wraps it into WETH, joins it through the Cropper into the VAT and locks it into a debt position.
- `lockGem()`: Allows users to deposit tokens, joins it into the VAT through the Cropper and locks it into a debt position.
- `lockETHAndDraw()` / `lockGemAndDraw()`: Allows a user to deposit, lock Ether or Tokens as collateral, generate debt and receive DAI.
- `freeETH()`: Allows users to unlock WETH in his debt position and receive it as Ether.
- `freeGem()`: Allows users to unlock collateral token in his debt position and have it transferred.
- `exitETH()`: Allows users to exit unlocked WETH collateral and receive Ether through `Cropper.exit()` that performs harvesting.
- `exitGem()`: Allows users to exit unlocked collateral through `Cropper.exit()` that performs harvesting.
- `fleeETH()`: Same as `exitETH()` but uses `Cropper.flee()` which does not harvest.
- `fleeGem()`: Same as `exitGem()` but uses `Cropper.flee()` which does not harvest.
- `open()`: Opens the cdp for `ilk` and `usr` on the CdpRegistry.
- `openLockETHAndDraw()`: Opens the cdp for `ilk` and `msg.sender` on the CdpRegistry and calls `lockETHAndDraw()`.
- `openLockGemAndDraw()`: Opens the cdp for `ilk` and `msg.sender` on the CdpRegistry and calls `lockGemAndDraw()`.
- `crop()`: Harvests rewards and sends them to the user.
- `draw()`: Allows users to withdraw DAI. Calculates the amount of additional debt needed and modifies the urn using `frob()`.
- `wipe()/wipeAll()`: Allows users to repay some or all of their debt.
- `wipeAndFreeETH()`/ `wipeAllAndFreeETH()`: Allows users to repay some or all of their debt and withdraw Ether as collateral.
- `wipeAndFreeGem()`/ `wipeAllAndFreeGem()`: Allows users to repay some or all of their debt and withdraw the collateral token.
- `hope()` / `nope()`: Allow another address to act on one's urnproxy in the Cropper contract.
- `quit()`: Executes `quit()` on the Cropper. Used when the VAT is shut down.

3.3.2 DssProxyActionsEndCropper

Also, the DssProxyActionsEndCropper was added in version 3, intended to be used in case the Vault Engine is shut down. It allows users to settle & retrieve their funds. For a detailed explanation of the shutdown process please refer to the documentation: https://docs.makerdao.com/smart-contract-modules/shutdown/end-detailed-documentation. This contract wraps functionality to facilitate interaction through vaults held at the Cropper through UrnProxys during shutdown.

- `freeETH()`: Frees the position through the End contract and transfers Ether to the user.
- `freeGem()`: Frees the position through the End contract and transfers Tokens to the user.
- `pack()`: Locks new DAI supplied from the DSProxy into a bag in preparation to the call to cashETH/Gem().
- `cashETH()`: Allows to withdraw the Ether for the DAI locked in pack().
- `cashGem()`: Allows to withdraw the collateral token for the DAI locked in pack().

Note that the functions for `cash()` use `flee()` instead of `exit()` to exit the collateral tokens which means that the caller is forfeiting the rewards. `flee()`, instead of `exit()`, has been chosen as the default since the accounting in CropJoin is not updated automatically during the shutdown process when `skim()` is executed. In that step, the VAT grabs and assigns it to the End. Updating the accounting of the rewards could be done manually by executing `CropJoin.tack()` but in general there is no incentive to call this function after every `skim()`. Hence, the `stake` of the End contract in the CropJoin contract is not necessarily updated and insufficient stake could be available to transfer onward after `end.cash()`. Users wishing to withdraw the rewards may do so outside of the proxy actions and handle this on a case by case basis.

3.3.3 Changes in version 4:

Function `quit()` has been removed from the regular proxy actions contract since the Cropper's `quit()` can only be used during shutdown.

4 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.

5 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.

6 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:

- 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.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Unnecessary Additional Transfers Acknowledged

6.1 Unnecessary Additional Transfers

Design Low Version 3 Acknowledged

In functions `exitGem`, `wipeAndFreeGem`, `wipeAllAndFreeGem` of the DssProxyActionsCropper as well as in function `freeGem` of the DssProxyActionsEndCropper the collateral is first exited to the DSProxy before being transferred onward to `msg.sender`. The gems may be exited to `msg.sender` directly.

Acknowledged:

MakerDAO replied:

```
This is intentional to keep the rewards in the ds-proxy account. This way the ds-proxy owner can
choose when to withdraw them. This paradigm of leaving the rewards in the ds-proxy is used for
all actions calling join and exit. It was preferred by some of the UI projects integrating with
Maker.
```

7 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	5

- Documentation Not up to Date Specification Changed
- Event Fields Consistency Code Corrected
- Incorrect Decimal Annotation Code Corrected
- Urn Proxy Load Inefficiency Code Corrected
- NewProxy Event Consistency Code Corrected

7.1 Documentation Not up to Date

Design Low Version 1 Specification Changed

README.md describes `tack` and the auction process:

> The winner of a collateral auction claims their collateral via *flip.deal*

This describes the old liquidation process using the Cat. The ilks of CropJoin however will use liquidations 2.0 with the Dog and a slightly altered auction contract. `Flip.deal()` is not part of this.

Other recent maker projects contained a "Risk" section in their documentation. Different than the traditional GemJoin adapter which just locks the collateral, CropJoin stakes the collateral into a third party system. This introduces new risks which should be documented appropriately.

Specification changed:

The readme was updated and now describes the liquidation process using the Dog. Risk consideration have not been added.

7.2 Event Fields Consistency

Design Low Version 1 Code Corrected

1. In `CropJoin`, the `Join` and `Exit` events differ from the system's default ones in the sense that they do not contain the indexed address field.
2. The `Flee` event has no fields. Emitting this event will cost gas and will not give any useful information off-chain.

Code corrected:

1. The events now include the addresses of the urn and the user.
2. The `Flee` event now features following fields: The addresses for the user and the urn, the amount.

7.3 Incorrect Decimal Annotation

Correctness Low Version 1 Code Corrected

The decimals of `bonus` and `total` could differ.

```
uint256     public share;  // crops per gem    [ray]
uint256     public total;  // total gems       [wad]
uint256     public stock;  // crop balance     [wad]
```

`Share` is calculated and updated in `CropJoin.harvest()`:

```
function harvest(address from, address to) internal {
    if (total > 0) share = add(share, rdiv(crop(), total));
    uint256 last = crops[from];
    uint256 curr = rmul(stake[from], share);
    if (curr > last) require(bonus.transfer(to, curr - last));
    stock = bonus.balanceOf(address(this));
}
```

`share` is in `ray`, `crop()` is decimals of `bonus` and `total` is `wad`. Note that `rdiv()` multiplies `crop()` with a ray. The resulting unit for share is `bonus decimals * ray`, bonus decimals may not be in 18 decimals. `rmul(stake[from], share);` correctly calculates the amount in bonus token unit. The calculations are correct but the unit annotations are not accurate.

Code corrected:

The annotations have been updated:

```
uint256     public share;  // crops per gem    [bonus decimals * ray / wad]
uint256     public total;  // total gems       [wad]
uint256     public stock;  // crop balance     [bonus decimals]

mapping (address => uint256) public crops; // crops per user  [bonus decimals]
```

7.4 Urn Proxy Load Inefficiency

Design Low Version 1 Code Corrected

The `flux` function of `Cropper` calls `getOrCreateProxy` to load the source urn proxy. Here a check similar to the one in `move` would save gas, firstly by saving a function call, and secondly by reverting the transaction earlier in case the urn does not exist. If the source urn does not exist, `tack` will revert on non-zero `wad` parameter.

Code corrected:

`Flux()` now loads the source urn proxy directly from the proxy mapping and reverts if no entry exists.

7.5 `NewProxy` Event Consistency

Design Low Version 1 Code Corrected

The event `NewProxy`, which is triggered when a new urn proxy is deployed, is implemented in the Charter contract but not in Cropper. Since the two contracts are meant to behave similarly, the `NewProxy` event should consistent.

Code corrected:

The `NewProxy` event has been added to the UrnProxy contract in Cropper.sol.

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 UrnProxy With Different `ilk`

Note Version 1

It is possible to send `gem` that do not correspond to any `ilk` managed by the `Cropper` to an urn proxy and generate debt from this collateral. Since the urn proxy is managed by the `Cropper` it is not possible to get this `gem` out of the urn proxy.

8.2 Withdraw Value on Flee

Note Version 1

The internal accounting inside CropJoin is based on shares with 18 decimals:

Upon join(), the amount of tokens brought is converted into shares by expanding the value to an 18 decimals representation and a division through the net value per share:

The opposite is done during `exit()`.

This amount of shares is used for all accounting purposes, namely to update the `gem` balance in the VAT, the `stake[urn]` and the `total`.

SynthetixJoin.flee() retrieves the value stored in the `gem` mapping at the VAT. This value is in the unit of `shares`. This value however is used as the amount of tokens to withdraw from the pool.

```
function flee(address urn, address usr) public override {
    if (live == 1) {
        uint256 val = vat.gem(ilk, urn);
        if (val > 0) pool.withdraw(val);
    }
    super.flee(urn, usr);
}
```

The unit mismatch (shares vs tokens) is not problematic for SynthetixJoin under the condition that the gem token has 18 decimals. Due to the asset valuation in SynthetixJoin (`nav()`) the exchange rate between token and shares is 1:1.

In general, for an arbitrary CropJoin contract this may not hold. It's problematic when the exchange rate is not 1:1 or the token doesn't have 18 decimals. If too little tokens are withdrawn from the pool the amount of tokens available at the Join adapter are insufficient to transfer the required amount to the user and the whole transaction fails blocking withdrawals. The other way, rounded surplus tokens will remain at the CropJoin contract.