PUBLIC

Code Assessment

of the DSS-Charter Smart Contracts

November 17, 2021

Produced for

by

Contents

1 Executive Summary

Dear Maker team,

First and foremost we would like to thank MakerDAO for giving us the opportunity to assess the current state of their DSS-Charter system. This document outlines the findings, limitations, and methodology of our assessment.

The code reviewed was of a high standard. However, no documentation for the smart contracts reviewed was provided for the audit.

No security issue has been uncovered during the audit. The findings include a medium severity design issue:

- Possible Revert Due to Underflow

Several low severity design issues and notes are in this report. For a complete list of issues please refer to the Findings section.

After the intermediate report all reported issues have been addressed except some optimization issues which have been acknowledged.

The communication with your team during the audit was very good and helped to resolve arising questions quickly.

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

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	1
• Code Corrected	1
Low-Severity Findings	6
• Code Corrected	3
• Acknowledged	3

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-Charter 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	18 October 2021	3ea76ac216216c24f26f374ba5d3809066026ade	Initial Version
2	16 November 2021	2a62443ee1fa5d37d5c0ea78862d145d53288c7b	After Intermediate Report

For the solidity smart contracts, the compiler version `0.6.12` was chosen. This version, although being deprecated, has been chosen explicitly to be consistent through all newly developed maker modules.

The files in scope for this review were: `CharterManager.sol`, `join-managed.sol` and `DssProxyActionsCharter.sol`.

2.1.1 Excluded from scope

All files not listed above.

2.2 System Overview

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

DSS-Charter introduces a permissioned vault manager which allows users to take debts with origination fees instead of standard fees of Maker (stability fee). This is targeted for institution which have off-chain agreements with Maker. The fee is accrued when debt is taken and in exchange those urns feature fix/beneficial lending rates. For this, special `ilks` (collateral types) will be enabled in the Vat of the Maker system. These `ilks` use a special join adapter, which is also part of this review (`join-managed`). The `join-managed` adapter ensures that entrance/exit of collateral happens through the CharterManager only, and the CharterManager contract ensures that this is done only for urnproxys.

The receiver of the fee in form of generated DAI is the VOW Contract (Settlement Engine).

Anyone may open an urnproxy at the CharterManager contract and deposit collateral in form of a supported ilk. Apart from permissioned vaults, un-permissioned vaults may be supported as well. Note that, by default the un-permissioned mode is enabled for any `ilk` where any user is allowed to draw debt. The mapping `gate` allows wards to enable the permissioned mode per `ilk`. For ilks with the permissioned mode enabled, only whitelisted accounts, namely accounts that have received a non-zero debt ceiling may draw debt. Attempts of un-permissioned vaults to draw debt for those `ilk` fails as their debt ceiling is zero.

Joining or exiting collateral and repaying debt (call to `frob()` with `dart` less or equal to zero) are indifferent between permissioned and unpermissioned vaults for any ilk.

The intended use is that each user executes the DssProxyActionCharter code through his own DSProxy. Note that it's nevertheless possible to directly interact with the CharterManager.

The contracts of the system are:

CharterManager:

The main manager contract for DSS-Charter functionality. User are expected to use the DssProxyActionsCharter contract executed in the context of their own DsProxy contract to interact with the CharterManager. Interacting directly with the CharterManager is possible but must be done with care as there are some limitations which could result in lost funds. For each user a urnproxy contract is deployed which is in control of the urn at the Vat. During creation, the urnproxy gives full permission to the CharterManager (`VAT.hope()`). This ensures that all interactions with the urns created through the CharterManager requiring permissions can only be executed through the CharterManager. Notably owners of a UrnProxy must not attempt to repay debt by either joining DAI into their UrnProxy or moving DAI into their UrnProxy using `VAT.move()`. DAI balance in the mapping at the VAT will be inaccessible. Repaying `dart` works by taking DAI from `msg.sender` vault at the VAT while executing `frob()` on the urnproxy.

The CharterManager features a mapping `proxy` to store the link between urnproxys and users.

Following functions are available to interact with the urn:

- `join`: Allows to deposit a supported collateral
- `exit`: Allows `msg.sender` to exit collateral from his urn
- `frob`: Allows to modify a vault if `msg.sender` has the permission to do so. Reducing the debt is not subject to any restriction. Increasing the debt however has certain conditions, these condition depend on whether the chosen ilk is restricted to permissioned vaults only or not:
    - For ilks Restricted to Permissioned Vault the fee parameter `nib` and the minimum health factor `peace` are set per user
    - Otherwise these parameters are set per ilk

Note that the extra health check may be disabled.

- `flux`: Allows to transfer collateral in the Vat accounting between users if `msg.sender` has the permission to do so.
- `quit`: In case of shutdown of the Vat, allows to move the `ink` and `art` balance to another urn, this urn may be a "free" urn not managed by the CharterManager.

The owner of the urnproxy (which is the DSProxy of the user if the user uses DssProxyActionsCharter as intended) can give and remove approval to other addresses to act on their behalf using the `hope/nope()` functionality.

Note that it is possible although useless to use the CharterManager to interact with normal or deprecated ilks.

Unhealthy urns created through the CharterManager may be liquidated. There is a caveat participants of the auction have to be aware of: The exit of the collateral bought via the managed-join adapter can only be done through the CharterManager and the CharterManager only operates on proxy vaults. Hence a user either supplies the address of his urn proxy in the auction or moves the gem using vat.flux to such a vault in order to be able to exit the gem via the CharterManager. Using the public `getOrCreateProxy` function of the CharterManager anyone may create an urn proxy.

The CharterManager is upgradable through a proxy scheme: The main CharterManager contract features functionality to add/remove wards and to set the implementation address. The fallback function executes the code of the set implementation address as delegatecall for all other function selectors.

Join-Managed:

Permissioned Join adapter to deposit/withdraw collateral. The permission is required to ensure all interaction, including depositing of collateral happens through the CharterManager.The CharterManager ensures that only `urnproxy` vaults can deposit.

DssProxyActionsCharter:

To facilitate the interaction with the core functionality a wrapper contract DssProxyActionsCharter exists. This is an adapted version of the ProxyActions contract.

The interaction with the CharterManager contract is intended to happen through the DssProxyActionsCharter contract as follows: Each user deploys his own DSProxy through which he executes the code of the DssProxyActionsCharter contract. Note that the DsProxy contract of the user is the owner of the UrnProxy in the CharterManager.

Following functionality are implemented:

- `lockETH()`: Allows users to send Ether, wraps it into WETH, joins it through the CharterManager into the Vat and locks it into a debt position.
- `lockGem()`: Allows users to deposit tokens, joins it into the Vat through the CharterManager 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.
- `exitGem()`: Allows users to exit unlocked collateral.
- `draw()`: Allows users to generate DAI given to available collateral and receive DAI
- `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 CharterManager contract.
- `quit()`: Executes `quit()` on the CharterManager. Used when the Vat is shut down.

DssProxyActionsEndCharter:

Used in case the Vault Engine is shut down. 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 Charter 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().

2.2.1 Trust Model & Roles

Wards: Fully trusted to behave honestly and correctly at all times. They can set the parameters and can update the implementation of the CharterManager contract.

Permissioned Users: Whitelisted for certain `ilks` with a set debt ceiling. The ceiling only applies if the `Gate` is set for this `ilk` to be used by permissioned vaults only.

Un-permissioned Users: Untrusted. Can interact with the CharterManager contract with un-permissioned ilks only.

Users must fully trust the CharterManager contract as it features 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 CharterManager.

Users are expected to use their own DsProxy and execute the code of the DsProxyActionsCharter contract.

We assume that the DssCharterProxyAction is used frequently and hence calls to jug.drip() are done frequently updating the ilks rate.

Tokens: The CharterManager disregards the return value of the ERC20 `transfer()`/`transferFrom()` and `approve()` calls. ERC-20 tokens that do not revert on failure are not supported. In the Maker system the adapter (join) will take care of the token idiosyncrasies and hold the collateral tokens. The join-managed 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 join-managed contract. Generally it is assumed that only trusted tokens are added to the Maker system. Finally we assume all tokens have no more than 18 decimals. Note that although the join-managed contract enforces this, older join adapters which are still in use did not enforce this restriction.

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 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 associated with the items defined in the engagement letter on whether it is used in accordance with its specifications by the user meeting the criteria predefined in the business specification. 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 the facts that over the life cycle of any software product changes to the product itself or to its environment, in which it is operated, can have an impact leading to operational behaviours other than 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 severities. 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:

- Design: Architectural shortcomings and design inefficiencies

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	3

- Inefficient _validate Acknowledged
- Skip Calls When No Additional Debt Is Needed Acknowledged
- getOrCreateProxy() or proxy[msg.sender] Acknowledged

5.1 Inefficient `_validate`

Design Low Version 1 Acknowledged

`_validate` may be refactored to be more efficient. The amount of external calls executed may be reduced.

By checking whether a check of the credit line or the peace is even required first, the call to the Vat and the calculation of the `tab` could be skipped in case it's not needed. The current code however calls the Vat initially and then calculates the `tab`, before determining whether a credit line or peace check is needed.

Acknowledged:

Maker acknowledged the issue.

5.2 Skip Calls When No Additional Debt Is Needed

Design Low Version 1 Acknowledged

`DssProxyActionsCharter.draw()` generates the debt required before exiting the DAI amount to the user's wallet:

```
// Generates debt in the CDP
_frob(charter, ilk, 0, _getDrawDart(charter, vat, jug, ilk, wad));
...
// Exits DAI to the user's wallet as a token
DaiJoinLike(daiJoin).exit(msg.sender, wad);
```

`_getDrawDart()` may return 0 if no additional debt is required to exit the specified amount of DAI. The calls to `CdpManager.frob()` and `Vat.frob()` will execute nevertheless in this case, despite not being required.

Acknowledged:

Maker acknowledged the issue.

5.3 `getOrCreateProxy()` or `proxy[msg.sender]`

Design | Low | Version 1 | Acknowledged

The CharterManager implementation has a function `getOrCreateProxy()` which returns the address of an urn managed by the CharterManager for a user, or creates a new urn if it does not exist yet. Although, the Charter Manager features a public mapping `proxy` which stores the list of urns and their respective users, multiple functions in the DssProxyActionsCharter use `getOrCreateProxy` function even when not necessary, i.e., there is no need to create a new urn if it does not exist already. Examples of such functions are `wipe()`, `wipeAll()`, `cashETH()`, or `cashGem`.

Similarly this also applies to `CharterManager.quit()`.

The CharterManager features functions `exit` and `flux`. Both operate on the collateral of the user in the Vat. While `flux` transfers the collateral in the accounting of the Vat to another address, `exit` exits the collateral to the user.

From an users perspective, for the account which is the source of the collateral these should behave similarly. `Exit()` however uses `proxy[msg.sender]` to load the address of the Urnproxy, while `flux()` uses `getOrCreateProxy(src)`.

Acknowledged:

Maker acknowledged the issue.

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	1

- Possible Revert Due to Underflow Code Corrected

Low-Severity Findings	3

- Inconsistent Retrieval of Ilk Parameter Code Corrected
- Possible Optimization on Getting vat Address Code Corrected
- Unused Function _toRad() Code Corrected

6.1 Possible Revert Due to Underflow

Design Medium Version 1 Code Corrected

Should the recorded DAI balance of the DSProxy at the Vat exceed the amount required to repay the debt, the subtraction in `DssProxyActionsCharter._getWipeAllWad()` will underflow causing the transaction to revert.

Code corrected:

`_getWipeAllWad()` now returns 0 when enough DAI is available to cover the debt.

6.2 Inconsistent Retrieval of Ilk Parameter

Design Low Version 1 Code Corrected

The function `cashETH` in the DssProxyActionsCharter contract takes as parameters `ethJoin` and `ilk` among others. However, in other functions, e.g., `freeETH()`, `wipeAllAndFreeETH()`, etc. only `ethJoin` is passed as parameter, while the `ilk` value is retrieved from the adapter: `bytes32 ilk = GemJoinLike(ethJoin).ilk()`.

Code corrected:

`cashEth()` and `cashGem()` now retrieve the `ilk` from the adapter as the other functions do.

6.3 Possible Optimization on Getting `vat` Address

Design Low Version 1 Code Corrected

Multiple functions in the DssProxyActionsCharter and DssProxyActionsEndCharter contracts receive the `vat` address as follows: `address vat = CharterLike(charter).vat()`. Considering that the `vat` contract is already deployed and its address is not expected to change, the contracts can store this value as immutable or constant to optimise gas costs.

Code corrected:

Both the address of the VAT and the CharterManager (which was previously passed as function argument) are now stored as immutables.

6.4 Unused Function `_toRad()`

Design | Low | Version 1 | Code Corrected

The function `_toRad()` is implemented in `DssProxyActionsCharter` but it is not used.

Code corrected:

The unused function has been removed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Overflow When Drawing More Than 100 Trillion Debt

Note Version 1

Theoretically, the function `_getDrawDart()` can overflow when computing dart: `dart = _toInt256( _mul(netToDraw, WAD) / _sub(_mul(rate, WAD), _mul(rate, nib)))`. `netToDraw` is in rad (45 decimals) and `wad` has 18 decimals, therefore for large `netToDraw` (greater than 10**14) the computation overflows.

7.2 Possible Overflow in `exit()`

Note Version 1

The function `exit()` in ManagedGemJoin contract converts `uint256 wad` into a negative value: `-int256(wad)`. Before the conversion, the following check is performed to prevent overflows: `require(wad <= 2 ** 255)`. Theoretically, if `wad == 2 ** 255` the overflow will happen twice, but the result matches the expected value in this case.

7.3 Unaware Users and Permissioned Ilks

Note Version 1

Unaware users may deposit collateral for a permissioned ilk. Only when a user attempts to draw debt for such a permissioned ilk the transaction will revert.

The reason is that for a permissioned ilk an unpermissioned user has a credit line of 0, hence cannot take on debt. The error message `CharterManager/user-line-exceeded` and the place where the transaction reverts may be confusing for an unaware user.