PUBLIC

Code Assessment

of the DSSProxyActions Smart Contracts

December 6, 2022

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

MakerDAO implements a new version of the proxy actions contract that, similar to the previous proxy actions contract, offers functions that batch interactions with the DAI Stablecoin system.

The most critical subjects covered in our audit are functional correctness and interactions with the core contracts. Security regarding all the aforementioned subjects is high.

The general subjects covered are code complexity, gas efficiency and error handling. Security regarding all the aforementioned subjects is high.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3
• Code Corrected	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 DSSProxyActions 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	11 November 2022	4624b701a4f38c4f62f07a8fde55eb1ce7c8532e	Initial Version
2	05 December 2022	8017ce143d75cd81bf36b4e02b2f34ec64164ba7	After Intermediate Report
3	06 December 2022	40bb4493e7cf4fac92f84e0b021d3bc44fb7b752	Final Version

For the solidity smart contracts, the compiler version `0.6.12` was chosen.

The file DssProxyActions.sol with contracts

DssProxyActions

DssProxyActionsEnd

DssProxyActionsDsr

was in scope of this review.

2.1.1 Excluded from scope

Contracts the Proxy Actions interact with, e.g.

CDPManager

ProxyRegistry

Pot

and core contracts of the DAI Stablecoin System such as VAT, JUG or END

as well as the Ds/DssProxy are not in scope of this review.

2.2 System Overview

The new proxy action contracts implement functions aggregating actions for interaction with the DAI Stablecoin system. These functions are intended to be executed through the user's proxy contract. This proxy may be a Ds-Proxy or Dss-Proxy.

Some of the functions of the DssProxyActions work on urns managed by the CDPManager. Such positions are identified by a CDP (collateralized debt position) id. An urnproxy is used to hold the position in the VAT. For more information please refer to the documentation of the CDPManager. Other functions

access the urn of the proxy or the funds held by the proxy directly. DssProxyActionsEnd exits positions from the CDPManager and thereafter handles them directly from the proxy context. DssProxyActionsDsr does not use the CDPManager.

In contrast to the previous version, immutables are now used to store addresses of system contracts, due to this the interface of some functions changed.

2.2.1 DssProxyActions

The following functionalities are implemented:

`transfer(address gem, address dst, uint256 amt)`: Transfers specified token from the source to the destination.

`ethJoin_join(address ethJoin, address urn)`: Payable function joining `msg.value` into the urn.

`gemJoin_join(address gemJoin, address urn, uint256 amt)`: Joins the amount of tokens into the urn. Tokens originate from `msg.sender`.

`hope(address addr, address usr)`: Approve an address for one's urn.

`nope(address addr, address usr)`: Remove approval to act on one's urn.

`open(bytes32 ilk, address usr)`: Opens a new position in the CDPManager.

`give(uint256 cdp, address usr)`: Transfer the ownership of a CDP in the CDPManager to the passed address.

`giveToProxy(uint256 cdp, address dst)`: Transfers the ownership of a CDP in the CDPManager to the proxy of the passed address. The passed `dst` address may already own a proxy in the registry or only if `dst` is not a contract a new proxy is deployed.

`cdpAllow(uint256 cdp, address usr, uint256 ok)`: Change allowance of `usr` on the cdp position in the CDPManager.

`urnAllow(address usr, uint256 ok)`: Change allowance status of `usr` on the urn in the CDPManager.

`flux(uint256 cdp, address dst, uint256 wad)`: Moves `gem` held by the urnproxy of the CDP in the CDPManager to the urn address specified.

`frob(uint256 cdp, int256 dink, int256 dart)`: Calls `CDPManager.frob()`, which eventually executes `frob` to modify the urns

`ink`/ `art` on the VAT, all address parameters will be the `urnproxy`.

`quit(uint256 cdp, address dst)`: Calls `CdpManager.quit`, which eventually executes `fork()` on the VAT. This allows to migrate the cdp (`ink` / `art`) into the given destination urn.

`enter(address src, uint256 cdp)`: Calls `CdpManager.enter()`. Allows importing an urn into the CDP. Uses VAT`.fork()` to migrate the `ink` / `art` between the urns.

`shift(uint256 cdpSrc, uint256 cdpOrg)`: Calls `CdpManager.shift()`. Allows shifting the `ink`/ `art` from one CDP to another.

`lockETH(address ethJoin,uint256 cdp)`: This payable function joins Ether and locks it as collateral (`ink`) into the CDP.

`lockGem(address gemJoin, uint256 cdp, uint256 amt)`: This function joins Gem tokens and locks them as collateral (`ink`) into the CDP:

`freeETH(address ethJoin, uint256 cdp, uint256 wad)`: Frees locked Ether collateral (`ink`) and exits it to the user.

`freeGem(address gemJoin, uint256 cdp, uint256 amt)`: Frees locked collateral (`ink`) and exits it to the user.

`exitETH(address ethJoin, uint256 cdp, uint256 wad)`: Exits free Ether collateral (`gem`) to the user.

`exitGem(address gemJoin, uint256 cdp, uint256 amt)`: Exits free collateral (`gem`) to the user.

`draw(uint256 cdp, uint256 wad)`: Exits the specified amount of DAI tokens. Draws more debt if needed.

`wipe(uint256 cdp, uint256 wad)`: Joins the specified amount of DAI. If the sender has access rights on the CDP in the CDPManager, wipes debt of the CDPs urn using all available DAI. If the sender doesn't have access rights on this CDP the debt of the CDP is reduced using the joined DAI amount.

`wipeAll(uint256 cdp)`: Wipes all debt of the CDP. The amount of DAI needed must be made available either already at the VAT or the Proxy.

The following functions with "safe" prepended to the name feature an additional check ensuring the `owner` of the CDP before calling their base function described above:

```
safeLockETH(address ethJoin, uint256 cdp, address owner)
safeLockGem(address gemJoin, uint256 cdp, uint256 amt, address owner)
safeWipe(uint256 cdp, uint256 wad, address owner)
safeWipeAll(uint256 cdp, address owner)
```

Furthermore, the following functions aggregating one or more previously described functionalities exist:

```
lockETHAndDraw(address ethJoin, uint256 cdp, uint256 wadD)
openLockETHAndDraw(address ethJoin, bytes32 ilk, uint256 wadD)
lockGemAndDraw(address gemJoin, uint256 cdp, uint256 amtC, uint256 wadD )
openLockGemAndDraw(address gemJoin, bytes32 ilk, uint256 amtC, uint256 wadD)
wipeAndFreeETH(address ethJoin, uint256 cdp, uint256 wadC, uint256 wadD)
wipeAllAndFreeETH(address ethJoin, uint256 cdp, uint256 wadC)
wipeAndFreeGem(address gemJoin, uint256 cdp, uint256 amtC, uint256 wadD)
wipeAllAndFreeGem(address gemJoin, uint256 cdp, uint256 amtC)
```

2.2.2 DssProxyActionsEnd

The shutdown of the Dai Stablecoin System happens in several distinct steps as described in detail in the documentation:
https://docs.makerdao.com/smart-contract-modules/shutdown/end-detailed-documentation

This proxy actions contract facilitates the interaction in the shutdown process and exposes the following function:

`function freeETH(address ethJoin, address end, uint256 cdp)` and

`function freeGem(address gemJoin, address end, uint256 cdp)`: Frees locked collateral (`ink`). Settles the outstanding debt of the urn (`art`) if needed, exits the ink from the UrnProxy controlled by the CDPManager into a free urn then withdraws these funds to the `msg.sender` .

`function pack(address end, uint256 wad)`: Joins the amount of DAI specified and packs them into `bag`.

`function cashETH(address ethJoin, address end, bytes32 ilk, uint256 wad)` and

`function cashGem(address gemJoin, address end, bytes32 ilk, uint256 wad )`: During the final step of the Shutdown process, allows the user to redeem DAI previously added to `bag` for the corresponding share of the specified collateral gem.

2.2.3 DssProxyActionsDsr

Dai holders can lock their DAI into the DAI Savings Rate smart contract at any time. Once locked, DAI continuously accrues to the user's balance, based on the current DSR set by the Maker governance.

This proxy actions contract exposes the following function:

`join(uint256 wad)`: Deposits the amount of DAI specified.

`exit(uint256 wad)`: Withdraws the amount of DAI specified.

`exitAll()`: Withdraws all DAI of the `msg.sender` in the DSR cotnract.

2.3 Trust Model & Roles

The code of the proxy actions is executed in the context of each user's proxy. The proxy actions code simply aggregates individual calls and may calculate input values. For interactions with the core DAI Stablecoin System, the CDPManager is used. For some interactions, the urn held by the Proxy itself may be used.

System contracts such as the CDPManager, ProxyRegistry and the contracts of the Dai Stablecoin System are fully trusted.

The user is untrusted.

All tokens are expected to be fully ERC-20 compliant, have 18 or fewer decimals and must not have any special behavior including but not limited to fees or rebasing.

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:

- 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	0

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	3

- Skim in _Free() Code Corrected
- Unused Function _sub() Code Corrected
- Use Defined Constant Code Corrected

6.1 Skim in _Free()

Design Low Version 1 Code Corrected

For `End.free()` to be successful, the proxy's `art` must be zero. However, the following code of `DssProxyActionsEnd._free()` does not strictly enforce that.

```
function _free(
    address end,
    uint256 cdp
) internal returns (uint256 ink) {
    bytes32 ilk = manager.ilks(cdp);
    address urn = manager.urns(cdp);
    uint256 art;
    (ink, art) = vat.urns(ilk, urn);
    // If CDP still has debt, it needs to be paid
    if (art > 0) {
        EndLike(end).skim(ilk, urn);
        (ink,) = vat.urns(ilk, urn);
    }
    // Approves the manager to transfer the position to the proxy's address in the vat
    if (vat.can(address(this), address(manager)) == 0) {
        vat.hope(address(manager));
    }
    // Transfers position from CDP to the proxy address
    manager.quit(cdp, address(this));
    // Frees the position and recovers the collateral in the vat registry
    EndLike(end).free(ilk);
}
```

First, in case that `art` is non-zero, `end.skim()` is executed on the urnproxy. Next, the position is transferred to the proxy. Note that the proxy could have non-zero `art`. Hence, the call `end.free()` which frees the collateral of its `msg.sender` and requires that `art` is zero could revert.

Code corrected:

Now, `skim()` is called for the proxy and after the CDP has been transferred from the urn. Thus, it is enforced that `art` will be zero.

6.2 Unused Function _sub()

Design Low Version 1 Code Corrected

The internal function `_sub` is never used.

Code corrected:

The unused function was removed.

6.3 Use Defined Constant

Design Low Version 1 Code Corrected

Contract Common defines:

```
uint256 constant RAY = 10 ** 27;
```

Instead of using the constant, function `_toRad` has this value hardcoded:

```
function _toRad(uint256 wad) internal pure returns (uint256 rad) {
    rad = _mul(wad, 10 ** 27);
}
```

Code corrected:

This function has been removed in the final version of the code reviewed.

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 Dust Amount of DAI to Be Drawn Leads to Revert

Note Version 1

There is a known issue in functions that use `_getDrawDart()`: In case the additional amount of DAI to be drawn leads to a dusty urn, the transaction reverts. This is a known edge case.

7.2 `dai` Shadowed

Note Version 1

Contract Common defines the immutable `dai`. The internal functions `_getDrawDart`, `_getWipeDart` and `_getWipeAllWad` each define a local `uint256 dai` which consequently shadows the immutable.

In Version 2 the variables in the functions have been renamed.