PUBLIC

Code Assessment

of the DSS Cure Smart Contracts

May 12, 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 DSS Cure according to Scope to support you in forming an opinion on their security risks.

Cure is an extension for the Dai Stablecoin System which allows contracts to report DAI amounts which must be subtracted from the total debt during the shutdown process. The necessity for this arose as a new extenstion, DSS-Wormhole generates such DAI which must not be included the settlement during shutdown.

The most critical subjects covered in our audit are security, functional correctness and the impact on the existing system. In summary, we find that the codebase provides a high level of security. There is a risk that the shutdown process is blocked in case the Governance pauses the Cure contract. For more information please refer issue description in this report.

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	1
• Risk Accepted	1
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 following source code files of the main DSS (DAI Stablecoin System) repository:

- cure.sol
- end.sol (new functionality related to Cure only)

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	4 April 2022	13654faec7efbc38a4d48f53130b31e00ac9e4be	Initial Version

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

2.1.1 Excluded from scope

All files not explicitly listed above.

2.2 System Overview

Cure adds functionality to the core DAI Stablecoin System which allows external sources to report DAI amounts that should be subtracted from the total DAI debt during the shutdown process. This is to account for special/locked DAI which is not eligible to and can't participate in steps 7 to 9 of the shutdown process, the equal distribution of the remaining collaterals between the remaining DAI holders. The first example for this is DSS-Wormhole: For DAI being bridged to L1 Ethereum, Wormhole already mints DAI to the user based on the promised DAI which eventually arrives via the slow bridge. The DAI minted based on the `ilk`/ collateral ("promised DAI") increases the VAT's debt until they are settled by the DAI arriving via the bridge. During shutdown this settlement can no longer take place and instead the bridged DAI will remain locked in the WormholeJoin contract. As these DAI are not free they must not and can't participate in the shutdown process. As they are accounted for in `vat.debt`, the shutdown process must remove them from the total debt existing in order for the calculation of the collateral distribution to be correct.

The core changes to the end contract to integrate the new cure functionality are:

- `cage()`: This function is called in step 2 of the shutdown process. It calls `cage()` on all components of the core DAI Stablecoin System, a call to `cure.cage()` has been added.
- `thaw()`: This function is called in step 6 of the shutdown process. It sets the final debt of the system. The new code now sets the value to `sub(vat.debt(), cure.tell())` instead of `vat.debt()`.

The new cure contract supports multiple sources to report amounts to be deducted from the total debt. While the contract is live (has not been caged yet), all state changing functions can only be executed by the privileged `ward` role (the governance).

The contract offers the following functionality:

1. Administrative functionality:

- `rely()`: Adds the given account as a `ward`.
- `deny()`: Removes the given account as a `ward`.
- `file()`: Updates the value for `when`. This value defines the timeout after which the shutdown process can continue despite a source not having reported an amount.

2. Cure functionality:

- `lift()`: Allows adding a new source. This source must implement the following interface:

```
interface SourceLike {
    function cure() external view returns (uint256);
}
```

- `drop()`: Allows removing a source.
- `cage()`: Cages the cure contract, this sets `live` to 0. Called by `end.cage()` during shutdown.
- `load()`: After the contract has been caged, anyone may execute this function and pass an existing source as parameter. Stores the value reported by the source. Updates the value when called again.
- `tell()`: Only when the cure contract has been caged and either all sources have reported their debt or the timeout has passed, returns the total amount reported by all sources. Used by `end.thaw()` to query the reported value.
- `tCount()`: Returns the number of sources.
- `list()`: Returns an array with the addresses of the sources.

2.2.1 Trust Model & Roles

Wards: For every contract, each address set to 1 in any of the wards mappings is fully trusted and expected to behave correctly. The governance and the end contract will bear this role.

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 our findings. 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	1

- Cure.cage() Might Block Shutdown Procedure Risk Accepted

Low-Severity Findings	1

- Possible Optimizations Acknowledged

5.1 Cure.cage() Might Block Shutdown Procedure

Design Medium Version 1 Risk Accepted

The `cage` function of the `Cure` contract, when called, requires that `live` is 1 and sets it to 0 :

```
function cage() external auth {
    require(live == 1, "Cure/not-live");
    live = 0;
    /*...*/
}
```

The function is meant to be called from the `End` contract :

```
function cage() external auth {
    /*...*/
    cure.cage();
    /*...*/
}
```

If an authorized user (the Governance) were to call the `cage` function of the `Cure` contract before the `End` contract, then `live` would be 0, therefore the call to `cage` would revert, effectively blocking the shutdown process.

Risk Accepted:

MakerDAO states:

```
We accept this risk as it is, and actually exists in other modules such as the Vow.
We understand each governance action might have important consequences. Each spell
needs to be carefully evaluated.
```

5.2 Possible Optimizations

Design Low Version 1 Acknowledged

When using a `uint256` as a boolean value, it is more efficient to check if it is non-zero than to check if it is equal to 1. Both the `auth` modifier and the liveness checks can be optimized in order to reduce gas costs and bytecode size.

The `auth` modifier could be implemented as follows:

```
modifier auth {
    require(wards[msg.sender] != 0, "Cure/not-authorized");
    _;
}
```

The liveness could be checked like this:

```
require(live != 0, "Cure/not-live");
```

In total these changes reduce the bytecode size by 36 bytes and the cost of each check by 6 gas.

Acknowledged:

Rather than prioritizing minimal gas optimization Maker prefers to follow the standard of how things have been done before.

6 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.

6.1 Delete Amt[Src] in Drop Function Is Useless

Note Version 1

In the `drop` function, the entry in the mapping `amt` that corresponds to the source that is removed is deleted :

```
function drop(address src) external auth {
    /*...*/
    delete amt[src];
    /*...*/
}
```

This function can only be executed when the `Cure` contract is live :

```
function drop(address src) external auth {
    require(live == 1, "Cure/not-live");
    /*...*/
}
```

On the other hand, the `amt` mapping can only be updated in the `load` function, which can only be executed when the `Cure` contract has been caged :

```
function load(address src) external {
    require(live == 0, "Cure/still-live");
    /*...*/
    uint256 newAmt_ = amt[src] = SourceLike(src).cure();
    /*...*/
}
```

Since the `Cure` contract cannot become live again after it has been caged, the `amt` mapping cannot be non zero during a call to `drop`, thus it is useless to remove the entry.