PUBLIC

Code Assessment

of Liquidations 2.0 Smart Contracts

April 16, 2020

Produced for

by

Contents

1 Executive Summary

First and foremost we would like to thank the Maker Foundation for giving us the opportunity to assess the current state of their Liquidations 2.0 system. This document outlines the findings, limitations, and methodology of our assessment.

Initially, our code assessment resulted in a number of findings regarding security and correctness. After the submission of the intermediate reports all findings have been resolved. These have been marked accordingly and can be found in the Resolved Findings section.

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

Yours sincerely,

ChainSecurity Team

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	4
• Code Corrected	2
• Specification Changed	2
Low-Severity Findings	6
• Code Corrected	5
• Specification Changed	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 general scope of the assessment is set out in our engagement letter with Maker Foundation dated February 5, 2020. The assessment was performed on the source code files inside the Liquidations 2.0 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	26 January 2021	c8a134429192a089bed6cc65ae8b73203fbb8374	Initial Version
2	4 March 2021	a4759e9d81ffb035fca5512a3f524e73923ac7bb	Version 2
3	21 March 2021	8ac41f0e6505352858deb88db9ac524346ac9e45	Version 3

The solidity smart contracts are expected to be compiled using compiler version `0.6.11`. After the intermediate report, the expected compiler version was changed to `0.6.12`.

In scope for this review is the new Liquidation 2.0 system only consisting of contracts defined in `dog.sol`, `clip.sol` and `abaci.sol`. Furthermore, the newly added `snip` function inside `end.sol` is in scope.

The documentation describing the changes for Liquidation 2.0 can be found at: https://github.com/makerdao/mips/blob/master/MIP45/mip45.md

The relevant documentation commit is: `3131664668f06b5046d052dcdf0b62fea9b66e69`

2.1.1 Excluded from scope

All contracts in files not explicitly listed above.

The correct choice of the parameters is out of scope of this review. A parallel engagement took place that was tasked to determine the correct parameters based on economic simulations.

3 System Overview

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

Liquidations 2.0 for multi collateral DAI has been developed to mitigate uncovered shortcomings in the previous liquidation system. The most notable change from the previous version is the move from English to Dutch style auctions. The resulting single block composability allows anyone to participate in the liquidation without capital constraints by leveraging flash-loans. Contrary to the old system, partial liquidations no longer exists except under special circumstances. Keepers, responsible to initiate the liquidation of undercollateralized vaults have no first mover advantage anymore in the auction, hence a new incentive scheme has been introduced.

3.1 Contracts

The liquidation 2.0 system consists of following deployed contracts: One shared `Dog` responsible for the vault liquidations and multiple `Clip` contracts handling the auctions, one for each collateral. Further contracts implementing the `AbacusLike` interface determine the price evolution during an ongoing auction.

3.1.1 Vault Liquidation - Dog.sol

Replaces the `Cat` contract of the old liquidation system.

In the context of the Maker protocol, a liquidation is the seizure of collateral from an insufficiently collateralized vault. During this process, the vault's debt is transferred to the protocol. An auction is started immediately to sell the collateral for DAI in an attempt to cancel out the debt now assigned to the protocol.

In `Dog.bark()`, the liquidation function takes all debt (different than before) and initiates the auction by calling `kick()` on the respective auction contract of this collateral type.

Several conditions must be met for a liquidation:

- The `Dog` contract must be `live`.
- The vault to be liquidated must be undercollateralized.
- The current liquidation amount of this collateral type must be below a certain limit.
- The global total liquidation amount of all collaterals must be below a certain limit.

  This prevents liquidation of too much collateral at once which could lead to price fluctuations of DAI. Additionally, allows to account for collaterals with limited liquidity.
- The auction contract for this collateral must accept new auctions.

The `Dog` contract has the following operational modes:

- `live == 1` - Status after deployment, system is active
- `live == 0` - System has been frozen after execution of `cage()`. No new liquidations can be initiated.

3.1.1.1 Special cases

Partial Liquidations: As the total amount of debt in active auctions cannot exceed either the global `Hole` or the per collateral `ilk.hole` a vault may only be liquidated partially.

Dust: Neither can vaults be left in a dusty state after a partial liquidation nor can auctions for partial liquidations be initiated below the dust amount.

In case the complete liquidation of the vault would exceed the `Hole/ilk.hole` limit but a partial liquidation up to the limit would leave a dusty vault - the `Hole/ilk.hole` limit is exceeded.

Liquidating a vault means:

1. Ensuring that the vault is undercollateralized by checking if the collateral amount multiplied with the collateral price accounting for the collateralization ratio is less than the debt consisting of the vault's `art` multiplied with the accumulated stability fee (`rate`).
2. The collateral amount (`dink`) and debt (`dart`) that will be removed from the vault are calculated. Normally these equal to `art` / `ink` except in the special cases mentioned above. In case the vault is only partially liquidated, the change in `ink` is proportional to the change in `art`.
3. `dart/dink` are removed from the vault's balances in the `Vat` contract, the vault engine. While the debt is assigned to the settlement engine `Vow`, the collateral is assigned to the auction contract.
4. The debt, `dart` multiplied with the stability fee (`rate`) is added to the system debt queue.
5. The `tab`, the target DAI amount the auction should recover, is determined by adding the liquidation penalty: The debt (`dart` multiplied with the stability fee) is multiplied with the liquidation penalty for this collateral.
6. An auction is started to recover this `tab` with the given collateral.

If the auction successfully recovers the `tab`, the settlement engine `Vow` receives the amount of DAI needed to cover the debt assigned in step 3 plus the liquidation penalty. The system features functionality to handle uncovered debt and surplus DAI which however are out of scope for this review of liquidations 2.0.

3.1.2 Collateral Auctions - Clip.sol

The `Clipper` contracts handle the auctions of the debt assigned to the protocol. Liquidations 2.0 implements Dutch style auctions where an auction starts at a high starting price, subsequently the price decays over time until a bidder accepts the offer. The proceedings of the auctions are transferred to the CDP engine.

One auction contract is deployed per `ilk` (collateral type).

The price decay is defined by an external contract implementing the `AbacusLike` interface exposing a `price(uint initial_price, uint seconds_since_auction_start)` function. Three such contracts currently exist, a `LinearDecrease`, a `ExponentialDecrease` and a `StairstepExponentialDecrease` contract.

Upon reaching a minimum price, an auction becomes inactive and reverts for every further bidding until restarted by calling `redo()`.

Bidders may buy partially from an auction.

The `Clipper` contract each have following operational modes:

- `stopped == 0` - Status after deployment, system is active
- `stopped == 1` - no new auction
- `stopped == 2` - no new start/restart of an auction or bid
- `stopped == 3` - no new start/restart of an auction or bid, no participation in auctions

The auction initiation function `Clipper.kick()` is called by the `Dog` upon initialization of a vault liquidation. Each auction has a unique auction id. An auction has the following parameters:

- Its position in the active auctions list
- Target amount of DAI to raise (`tab`)
- Amount of collateral available for purchase (`lot`)
- The address of the vault that was liquidated

- Timestamp of auction creation
- Initial price of collateral (`top`) determined by the price returned of the oracle (called `OSM`) plus a margin. This ensures auctions start at/above market price.

Anyone can participate in an ongoing auction and place a bid through `Clipper.take()`. The following parameters have to be specified:

- `id`: The id of the auction to bid on
- `amt`: The upper limit of collateral one is willing to buy
- `max`: The maximum price (DAI / collateral) one is ready to pay
- `who`: Receiver of collateral, receiver of external called
- `data`: Calldata, if length > 0 `who` is called from within the auction after the transfer of the collateral / before collection of the `DAI` amount due.

Note that the state of the auction one intends to participate may change between the signing of the transaction and the point in time when the transaction is actually executed. Furthermore, the price depends on the timestamp of the block the transaction has been included in. Hence estimating the precise outcome of the transaction in advance is hard.

Following scenarios can happen when bidding on auctions:

1. **Settling the tab while buying the full collateral up for sale** In this case the auction simply terminates.
2. **Settling the tab while buying only a part of the collateral up for sale** The auction terminates and the leftover collateral is assigned back to the liquidated vault.
3. **Settling the tab only partially for a part of the collateral up for sale** The auction remains active with the remaining amount of tab and collateral.
4. **Settling the tab only partially while buying the full collateral up for sale** The auction terminates. Note that the tab consists of the debt to be recovered and the liquidation penalty.

If less than the debt amount was recovered bad debt remains in the system. As mentioned above it can be settled by surplus DAI or debt auctions.

Special case `yank()`:

The `yank()` function allows privileged accounts to terminate auctions. This function is intended to be called by `end.snip()`. It terminates auctions and returns the collateral & debt to the vault. Note that the debt returned to the vault contains the liquidation penalty.

3.2 Roles and Trust Model

In this section we describe the roles and the trust model.

- `Wards`: Privileged accounts. They can execute functions with the `auth` modifier. This access control list exists separately in each deployed contract. A ward can add/remove other accounts for this role, set parameters or initiate privileged actions. Other system contracts requiring this privilege or the governance hold this role.

  These accounts are fully trusted to act honestly and correctly at all times. They can be seen as superusers as there is no concept of least privilege.

Roles of other contracts within system:

- `Price Oracle`: OSM, oracle security module. Returns the exchange rate with a delay of one hour. Note that the new auction system is dependent on the price oracle to reset the auction. Incorrect oracle prices could potentially liquidate the entire Maker ecosystem. Hence, oracles are extremely trusted.
- `CDP Engine`: Vault engine, called `vat`. Keeps track of all balances.

- `Settlement Engine`: Called `vow`. Handles `Debt` and `Surplus` DAI assigned the system.
- `Spotter`: The collateral price module called `spot`. Use to return the pricefeed of the collaterals.

All system contracts not part of this review are expected to work correctly at all times. In particular it is assumed that no reentrancies are possible through any calls to system contracts. This allows includes multi-layered calls, as is the case for the `Spotter` which returns the `Pip` address, that is later called. All of those addresses are considered trusted.

It is assumed that the accumulated rate of the stability fee for the collaterals is updated regularly. This is especially important for `Dog.bark()`, where the debt is calculated based on the current value of the rate for the collateral. An outdated value may be preferable for the vault's owner as a lower stability fee has to be paid. The keeper has an incentive to update the rate as it might push vaults past the collateralization ratio and as it potentially increases his reward. Note that there are also other incentives within the Maker system to update the rate, which can be done by anyone using `Jug.drip()`. One example is that vault owners are incentivized to update the rate by calling `Jug.drip()` before taking out a debt.

Following implicit roles exist in the system, although they have no special privileges:

`Keepers`: Accounts initiating Vault Liquidations / calling redo on unsuccessful auctions. They get compensation by the protocol if governance has set either `tip` or `chip` to a non-zero value or may be compensated otherwise.

`Bidders`: Any account may take part in an auction.

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 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.

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 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.

6 Findings

In this section, we describe any open findings. Findings that have been resolved, have been moved to the Resolved Findings section. All of the findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	0

Version 1

Spacer 0 20

Yours sincerely,

ChainSecurity Team

6.1 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	4
• Code Corrected	2
• Specification Changed	2
Low-Severity Findings	6
• Code Corrected	5

• Specification Changed	1

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	4

- Incorrect Description of Dog.bark() Specification Changed
- Dirt Remains After Bad Auction Code Corrected
- Potential Reentrancy During Emergency Shutdown Code Corrected
- Specification Mismatches Specification Changed

Low-Severity Findings	6

- Specification Mismatches Due to Final Changes Specification Changed
- Dust Retrieval Is Relatively Expensive Code Corrected
- Duplicate Functions in Clipper Code Corrected
- Gas Inefficiency During Auction Removal Code Corrected
- Liquidation of Dusty Vaults Code Corrected
- room > 0 Check Can Be Omitted Code Corrected

7.1 Incorrect Description of `Dog.bark()`

Correctness Medium Version 2 Specification Changed

After the intermediate report the main functions of Liquidations 2.0 have been annotated with their expected behavior taken from MIP45.

The description above `Dog.bark()` as well as the corresponding part in MIP45 are outdated: In order to address an other issue (Liquidation of Dusty Vaults), the behavior has been slightly altered.

Notably, the statement

```
// There is a precondition about `room` that needs
// to be satisfied in order to create an auction:
// room > 0 && room >= ilk.dust
// otherwise the transaction fails
```

no longer applies in the updated code.

Specification changed:

The code comments have been changed and now explain the new liquidation behaviour including the preconditions.

7.2 Dirt Remains After Bad Auction

Correctness | Medium | Version 1 | Code Corrected

As described in MIP45c4, the `Hole`/ `ilk.hole` values define a global / per-collateral limit of the total amount of `DAI` needed to cover the summed debt and liquidation penalty associated with all active auctions.

The current debt is tracked by the global `Dirt` and the per collateral `ilk.dirt` variables.

Upon auction initiation, the `tab`, the new debt of the system is added to the corresponding variables. Upon buying from an auction, the `owe` amount, the amount of debt paid, is removed from the corresponding variables.

The expected behavior is only loosely covered in MIP45c8:

```
Lastly, various values are updated to record the changed state
of the auction: the DAI needed to cover debt and fees for outstanding
auctions, and outstanding auctions of the given collateral type, are
reduced (via a callback to the liquidator contract) is reduced by owe,
and the tab (DAI collection target) and lot (collateral for sale) of
the auction are adjusted as well. If all collateral has been drained
from an auction, all its data is cleared and it is removed from the
active auctions list. If collateral remains, but the DAI collection
target has been reached, the same is done and excess collateral is
returned to the liquidated Vault.
```

As described in the specification above, the code only removes the received amount of DAI (`owe`) from the debt. This works as expected when the auction managed to cover the tab. In this case all debt added to the `dirt` during liquidation is removed. During exceptional circumstances however, the situation that an auction is unable to collect enough `DAI` to cover the `tab` despite selling all collateral may arise. In this scenario the auction terminates but the unrecovered debt amount remains in the `dirt` variables.

The expected behaviour in this scenario should be documented.

After such an auction, the value of `Dirt` will exceed the summed debt of all active auctions and it is no longer possible for the summed debt of all auctions to reach the limit defined by `Hole`.

If this happens repeatedly, e.g. during a rapid market crash the accumulated unaccounted dirt may severely restricts the amount of active auctions possible. Most notably this will impact less liquid collateral types with a comparatively low amount set for `ilk.hole`.

Code corrected:

The code has been updated and now handles this case correctly: When an auction has sold all collateral (`lot` reduced to `0`) the remaining `tab` is removed in addition to `owe` which is the aumount of DAI just collected:

```
// Removes Dai out for liquidation from accumulator
dog_.digs(ilk, lot == 0 ? tab + owe : owe);
```

7.3 Potential Reentrancy During Emergency Shutdown

Security | Medium | Version 1 | Code Corrected

Once the emergency shutdown mode has been entered, a reentrancy attack is possible. The reentrancy attack works as follows:

1. The attacker identifies an open auction to attack and the corresponding `ilk`. Note that this attack can be repeated for different auctions.
2. The attacker ensures that the `ilk` has been caged inside the `End` contract. If not, the attacker can enforce this by calling the `cage` function for the `ilk` in question.
3. The attacker calls `take` on the corresponding `Clipper` for the identified auction so that the auction will be closed at the end.
4. The attacker specifies its contract as `who` for the callback.
5. The `take` function sends collateral to `who`.
6. As part of the callback the attacker calls the `snip` function of the `End` contract, which will call the `yank` function of the `Clipper`.
7. The `snip` function returns the collateral and the debt to the corresponding vault . Hence, the collateral has been sent away twice at this point.
8. The `yank` function signals to the `dog` that the auction is closed. The `yank` function deletes the auction and removes it from the `active` list.
9. After the return of the callback, the take function also signals to the `dog` that the auction is closed, also deletes it and finally tries to remove it from the `active` list. At this point it removes another auction from the `active` list.

The consequences are:

- There are more active auctions than listed inside the `active` array.
- Not all remaining auctions can be closed. The `_remove` function will eventually revert once the `active` array is empty.
- The `Dirt` values of the `Dog` is incorrect. Hence, even auctions for other collaterals could revert during `yank` or `take` as the corresponding calls to `dog.digs` will revert.
- The `Clipper` does not hold sufficient collateral to serve all ongoing auctions.

Note that the exact consequences increase if the attack is performed multiple times.

Code corrected:

The issue was addressed by adding the `lock` modifier to `yank` which prevents the reentrancy.

7.4 Specification Mismatches

Correctness | Medium | Version 1 | Specification Changed

There are multiple errors of different severity in the MIP45 specification. For each item we list the relevant part of the specification and the explanation of the error:

- c7: "all liquidations disabled(2): This means no new liquidations (Clipper.kick), no takes (Clipper.take), and no redos (Clipper.redo)"

  Reason: While this is correctly implemented, the code comment is a bit unclear as it does not specify that no `kick` invocations are allowed on level 2:

```
// Levels for circuit breaker
// 0: no breaker
// 1: no new kick()
```

```
// 2: no new redo() or take()
```

- c8: "If the auction reached the tail value, ... then the Clipper.take would revert if called"

  Reason: This description of the `tail` value mismatches with its description in c1: "Time elapsed before auction reset". Note that the source code follows c1:

```
function status(uint96 tic, uint256 top) internal view returns (bool done, uint256 price) {
    price = calc.price(top, sub(block.timestamp, tic));
    done  = (sub(block.timestamp, tic) > tail || rdiv(price, top) < cusp);
}
```

- c8: "If the auction ... fell by cusp percent of top, then Clipper.take would revert if called, ..."

  Reason: Discrepancy with c1 "cusp = 0.6 * RAY (60% of the starting price), then the auction will need to be reset when reaching just below the price of 720." c1 implies that the auction needs to be restarted once it falls by at least `cusp` percent, while c8 implies that it needs to be restarted when it falls by more than `cusp` percent.

- c8: "If the caller provided a bytestring with greater than zero length, an external call is made to the who address, assuming it exposes a function, follow Solidity conventions, with the following signature."

  Reason: This is not entirely correct, as no call will be made if `who` is the `Dog` contract or the `Vat` contract.

- c13: "treats price at the current time as a function of the initial price of an auction and the time at which it was initiated".

  Reason: The price is a function of the initial price and the duration since last redo.

- c14: "This process will repeat until all collateral has been sold or the whole debt has been collected"

  Reason: This is not true as the auction might also be completed through a call to the `snip` function.

- c15: "The Clipper.take call can send any remaining collateral or DAI beyond owe to a cold wallet address inaccessible to the keeper."

  Reason: This statement is slightly imprecise as the remaining collateral or DAI would be moved by the `clipperCallee`.

- c16: "A mutex check to ensure the Clipper.take function is not already being invoked from `clipperCallee`."

  Reason:. The mutex check prevents reentrancy into `Clipper.take/redo()` irregardless of the `clipperCallee`.

- c17: "calls dog.digs in order to increment its Hole and ilk.hole values by the remaining auction tab."

  Reason: It is not Hole/hole that are modified but Dirt/dirt.

- c18: "function file(bytes32 what, uint256 data) external"

  Reason: `data` should be of the type `address`.

- c18: `function active() external view returns (uint256[]);`

  Reason: The automatically created getter `active` will requires numeric index as a parameter and returns a single `uint256`.

- c26: "urn.art * ilk.rate * ilk.chop ||"

  Reason: Missing operator for comparison.

- c26: In equations it seems that the units are not taken into account e.g., `urn.art * ilk.rate * ilk.chop > room`. However, this choice is not explicitly stated which creates mismatch with the implementation.

- c26: "vault.art * ilk.rate <= room"

  Reason: Missing `chop`.

- c27: "if amt < lot && tab - (amt * abacus.price) < ilk.dust"

  Reason: Mismatch with code. The code says `amt < lot && owe < tab`.

Specification corrected:

The specification has been corrected and matches the code behavior apart from minor diversions that are irrelevant to general usage, e.g., internal restrictions on callback targets.

7.5 Specification Mismatches Due to Final Changes

Correctness Low Version 3 Specification Changed

The documentation still lists the function `getId` inside the `Clipper` contract. However this function was removed in Version 3 of the code. Note that no functionality was removed as the `active` function can be used instead.

The `updust` function was newly added to the code in Version 3 to allow a caching of the `dust` value inside the `Clipper`, see Dust Retrieval Is Relatively Expensive for more information. The `updust` function is not yet documented.

Specification changed:

The specification was adjusted accordingly to reflect the removal of the `getId` function and the addition of the `updust` function.

7.6 Dust Retrieval Is Relatively Expensive

Design Low Version 2 Code Corrected

Multiple clever gas optimizations have been performed by the developers. However, we note that a rather trivial looking line still contains major gas costs:

```
(,,,, uint256 dust) = vat.ilks(ilk);
```

This line retrieves the dust amount for the specific ilk. It occurs inside the `Clipper` functions `take` and `redo`. Inside the `vat` the following struct will be loaded:

```
struct Ilk {
    uint256 Art;   // Total Normalised Debt     [wad]
    uint256 rate;  // Accumulated Rates         [ray]
    uint256 spot;  // Price with Safety Margin  [ray]
    uint256 line;  // Debt Ceiling              [rad]
    uint256 dust;  // Urn Debt Floor            [rad]
}
```

Hence, 5 SLOAD operations are necessary. After the activation of the upcoming `Berlin` hardfork this will cost 5 * 2,100 = 10,500 gas. However, in the current architecture there is no way to retrieve the `dust` value separately. Mirroring it inside the `Clipper` contract would reduce the costs significantly, but would introduce potential inconsistencies between the two values.

Code corrected:

The code has been corrected. The `dust` value is cached inside the `Clipper` and can be kept consistent through a permissionless call to `updust`.

7.7 Duplicate Functions in Clipper

Design | Low | Version 1 | Code Corrected

The two functions `getId` and the `active` inside the `Clipper` have the same functionality. Both functions take a list index as input and return the element of the `active` list at that index.

```
function getId(uint256 id) external view returns (uint256) {
    return active[id];
```

Hence, it seems that the code size is unnecessarily increased.

Code corrected:

The `getId` function was removed from the `Clipper` contract.

7.8 Gas Inefficiency During Auction Removal

Design | Low | Version 1 | Code Corrected

When an auction is being removed from the `Clipper`, because it is finished or has been yanked, then the auction id will be removed from the `active` list. As part of this removal, the auction id is exchanged with the last auction id in the `active` list and the appropriate changes are made:

```
function _remove(uint256 id) internal {
    uint256 _index   = sales[id].pos;
    uint256 _move    = active[active.length - 1];
    active[_index]   = _move;
    sales[_move].pos = _index;
```

In case that the removed auction was already last in the list, which is not unlikely given that there is such a list for each collateral, two SSTORE and one SLOAD operation could have been skipped.

Code Corrected:

The code has been changed as follows:

```
function _remove(uint256 id) internal {
    uint256 _move    = active[active.length - 1];
    if (id != _move) {
```

```
        uint256 _index   = sales[id].pos;
        active[_index]   = _move;
        sales[_move].pos = _index;
    }
    active.pop();
    delete sales[id];
}
```

7.9 Liquidation of Dusty Vaults

Correctness Low Version 1 Code Corrected

According to the protocol, the initiation of an auction will be reverted if the available `room` is less than the `dust` for the corresponding `ilk`. However, there might be the corner case where a dusty vault exists, e.g., after the `dust` amount for a particular `ilk` has been increased.

Dusty vaults can be blocked from liquidation even though there would be `room` for them. This is because of following check:

```
require(room > 0 && room >= dust, "Dog/liquidation-limit-hit");
```

Even though there wouldn't be enough room for dust, there would still be enough room for a dusty auction. This state is temporary. Later, once even more room becomes available again, the dusty vault can be liquidated again.

Code corrected:

Dusty vaults can now be liquidated. If there is `room` to liquidate the total `art` of the vault there are no further restrictions related to the `dust` to start the auction.

7.10 `room > 0` Check Can Be Omitted

Design Low Version 1 Code Corrected

In `dog.bark` there is a check on whether the available `room` is positive. This check takes place in the following require statement:

```
require(room > 0 && room >= dust, "Dog/liquidation-limit-hit");
```

This check is only useful in the case where `dust == 0` and `room == 0`, otherwise it holds trivially. In the previously mentioned case, however, `dart == 0` (since `dart = min(art, 0)`) and `(art - dart)*rate >= dust` (since `dust == 0`).

Hence, `dink = mul(ink, dart)/art == 0` and the following require statement reverts:

```
require(dink > 0, "Dog/null-auction");
```

Hence, the `room > 0` sub-condition can be safely removed, which saves a small amount of gas during every execution.

Code Corrected:

In the updated implementation the logic determining whether a full or partial liquidation happens has been changed in order to address another issue. The require statement listed above no longer exists and neither does an unnecessary `> 0` check. Hence the issue has been resolved.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The contracts in scope for this review are part of the Maker system which consists of many interacting contracts. Hence, the mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

8.1 Blocked Calls From Clipper

Note Version 2

During the `take` function of the `Clipper`, an external call is executed. Certain call targets are blocked:

```
if (data.length > 0 && who != address(vat) && who != address(dog_)) {
    ClipperCallee(who).clipperCall(msg.sender, owe, slice, data);
}
```

As the `Clipper` has special privileges inside the `vat` and `dog_` contracts, these contracts are blocked. However, additionally targets need to be blocked where the funds controlled by the `Clipper` could be moved in an unauthorized way. A good example is the `GemJoin.exit` function. This function could remove the stored collateral from the `Clipper` and send it to an attacker. Please note that this attack currently does not work as there is no collision between the signature hashes of

- `clipperCall(address,uint256,uint256,bytes)` and
- `exit(address,uint256)`.

However, we note that for all future contracts added to the system it needs to be ensured that no such collisions exist or the call targets need to be blocked.

8.2 Creation of Many Small Auctions

Note Version 1

As discussed in the MIP45, incentive farming needs to be avoided. Besides the scenarios that are already described in the MIP45, another scenario is possible. This scenario comes into effect if either the capacity for one collateral or the overall capacity has almost reached its limit. In technical terms this means that the `dirt` is almost as large as the `hole`.

If, in this scenario, a large vault becomes unsafe, an attacker could create many small auctions out of it. The attacker would perform the following steps within a single transaction:

1. Create a small auction that fills up the capacity limit and receive the keeper incentive
2. Take a small amount (ideally `dust`) from another auction (note that given that the limit is reached, there is likely a good auction available)

The only downside for the attacker compared to creating a big auction are higher gas costs. Note, however, that after EIP-2929 (in combination with EIP-2200) will come into effect the additional costs of performing step 2 multiple times will be significantly reduced, while the costs of repeated executions of step 1 will also be reduced.

8.3 Debt Queue Not Updated Automatically

Note Version 1

The `Vow` contract manages a system debt queue called `sin`, not to be confused with the `sin` mapping inside the `Vat` contract. It is noteworthy that the debt queue is fully not synchronized with the liquidation system. In particular, the liquidation system makes new entries, but never resolves them.

This can have two possible effects:

1. The debt inside the system debt queue is released too quickly. In particular that means that auctions might still be ongoing for the released debt and hence some of the debt might still get covered. This can occur if the `wait` value inside the `Vow` is too low in comparison to auction durations. As a consequence it might be possible to trigger a debt auction even though there is no need for it.
2. The debt inside the system debt queue is released too slowly. In particular that means that auctions might have long finished and that the debt has already been repaid. This can occur if the `wait` value inside the `Vow` is too large in comparison with auction durations. As a consequence surplus auctions could be unnecessarily delayed.

8.4 Ethereum Is a Dark Forest

Note Version 1

`Ethereum is a Dark Forest` describes the phenomena of bots inspecting the pool of unmined transactions and front-running profitable transactions with their own. Although the exact capabilities of these bots are unknown, these bots are sophisticated.

Liquidations 2.0 relies on `keepers` to initiate liquidations of undercollateralized vaults. There is a certain cost overhead (e.g. running a software monitoring the blockchain) for keepers to detect undercollaterlized vaults. Only after undercollateralized vaults have been identified, they can be liquidated by calling `Dog.bark()`. For their efforts, keepers are rewarded on-chain if `tip` and/or `chip` are set to non-zero values.

While it doesn't matter for the liquidation system when bots copy and front-run these transactions, the honest keepers will not only lose their anticipated reward for the liquidation, but also lose the gas fee paid. If this happens repeatedly, keepers may stop to identify & liquidate undercollateralize vaults as they can't make a profit. Once no keeper identifies and crafts transactions to liquidate vaults bots can't copy these transactions anymore - and hence in an extreme scenario no more liquidations happen.

`Clipper.redo()` is affected in a similar way, `Clipper.take()` may be affected partially, e.g. when there are flash-loans involved.

8.5 Incentive Farming Might Be Possible Due to Misconfiguration

Note Version 1

As mentioned in MIP45c19, Incentive Farming is a risk in the system. However, it could also occur without a change in the `dust` value. Note that there is no mechanism inside the smart contracts that prevents that a keeper's reward for kicking off an auction is bigger than the liquidation penalty which the system achieves. Hence, the governance needs to chose the corresponding parameters: `chip`, `tip` and `chop` very carefully as a misconfiguration allows a way to drain the system.

8.6 Initialization and Deployment Requires Extra Care

Note Version 1

As with any smart contract care needs to be taken during deployment and initialization. However, for these contracts it is especially important as they:

- will be integrated into an existing system
- are not fully initialized during deployment

In particular the following steps need to be performed correctly:

- Authorizations between the contracts need to be granted
- All parameters need to be chosen. Not that some functionality will already be available with partially initialized contracts, e.g. the Clipper contract will be fully functional if no `Vow` contract has been registered. However, all collected DAI will flow to the Zero address.
- Initially given deployment authorizations need to be revoked
- Authorizations for replaced contracts need to be revoked

8.7 Monotonicity of Price Functions

Note Version 1

The auction system is designed as "Dutch style auction system, where auction prices generally start high and drop over time". Note, however that there is no guarantee in the system that prices will be monotonically decreasing. Apart from a `redo` which can trigger a price increase, the prices can also rise due to changed parameters of the corresponding `Abacus` contract.

As an example, if the variable `tau` which contains the "Seconds after auction start when the price reaches zero" is increased, ongoing auctions will see a price increase. Note, that users of the system can protect themselves. Auction takers can specify a `max` price which they are willing to pay for collateral. Then, they only stand to lose gas costs.

We aim to educate users to properly use the `max` value even though there is a seemingly decreasing price.

8.8 No Stability Fee During Auctions

Note Version 1

As debt accumulates no stability fee during auctions, it needs to be ensured that debt doesn't reside inside an auction for too long. In case the stability fee would be very high, the liquidation penalty would be really low, and the auction would be running for a very long time, the stability fee lost during the auction time would exceed the liquidation penalty earned. In this hypothetical scenario a liquidation would be "beneficial" for a vault owner.

Note, however, that we deem this as highly unlikely as there is a general incentive to keep auctions short, which is also discussed in the MIP45. Even for the collateral with the currently highest stability fee (50%), the auction would have to take roughly 110 days to offset a regular liquidation penalty of 13%.

8.9 `Vat` Debt Tracking Not Automatically Synchronized

Note **Version 1**

At the beginning of an auction `dog.bark()` calls `vat.grab()` to reassign the collateral to the auction contract and the debt from the vault to the system. Hence, both `vat.sin[vow]` and `vat.vice` are increased by `dart` times the collateral's rate. The `sin` mapping and `vice` are used to track the bad debt of the system inside the `Vat` contract.

However, these values are not updated after a successful auction. This is due to how the Maker system works: After a purchase in an auction, the DAI amount received is transferred to the `vow`. When the `vow` contract has a surplus amount of DAI, anyone may call `vow.heal()` to settle the debt accrued in `vat.sin[vow]`. Further functionality allows to handle debt or surplus auctions. Please note that the `Vow` and `Vat` contracts are not in scope of this review and are expected to work correctly.