PUBLIC

Code Assessment

of the Claim Fee Maker Smart Contracts

May 12, 2022

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 8
4 Terminology 9
5 Findings 10
6 Resolved Findings 12
7 Notes 18

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 Claim Fee Maker according to Scope to support you in forming an opinion on their security risks.

Claim Fee Maker implements an addition to the Maker protocol enabling fixed-rate debt over a certain period of time. This addition works with existing ilks/vaults without the need for any change to the core system.

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

Claim Fee works by issuing claims for which the holder can claim compensation for the stability fee accrued. DAI for payout might be generated by minting unbacked stablecoin accounted to the VOW. Issuance collects no payment, the privileged role issuing claims must compensate the VOW accordingly, this is not handled by the smart contracts reviewed.

A claim fee is not connected to an actual debt position / urn. Plans exist to address this, please refer to note: No connection between ClaimFee and actual Debt.

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	1
• Code Corrected	1
Medium-Severity Findings	5
• Code Corrected	2
• Specification Changed	2
• Risk Accepted	1
Low-Severity Findings	9
• Code Corrected	6
• Risk Accepted	2
• Acknowledged	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Claim Fee Maker repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Claim Fee Maker

V	Date	Commit Hash	Note
1	02 Mar 2022	25f54df55b8f74311a17dc331c221a54b688afd6	Initial Version
2	02 May 2022	76dab7ab3f48efac325f092d2085c5f8415f0904	Second Version
3	11 May 2022	4c02193cc671f534ad85dbc2ae114ea93b32901b	Third Version

DSS Gate

V	Date	Commit Hash	Note
1	02 Mar 2022	71ea33e6868220aa2153daeb9e8134cc0c8478d9	Initial Version
2	25 Apr 2022	43260bfc5da0818d80bf6607485b9e74cbce9180	Second Version

The solidity smart contracts are written for compilers of versions `0.8.x`. The exact version has not been specified in the initial version reviewed. In Version 3 the compiler has been fixed to `0.8.13`, the most recent version at the time.

The files in scope are `ClaimFee.sol` of the Claim Fee Maker repository and `gate1.sol` of the DSS Gate repository.

2.1.1 Excluded from scope

All files not explicitly listed above.

2.2 System Overview

Claim Fee Maker implements an addition for the Maker protocol enabling fixed-rate debt over a certain period of time. To achieve this, a claim fee balance can be issued for a certain ilk (collateral type), issuance time and maturity. This claim fee balance can later be redeemed to offset the stability fee being accrued by a regular vault. The DAI amount required is generated by minting unbacked DAI in the VAT on behalf of the VOW. This concept allows implementing fixed rate debt for existing ilks/vaults with rates based on the stability fee without the need for any change to the core system.

The system consists of two contracts.

1. ClaimFee

Privileged actions to be executed by the governance:

- `issue`

    - Issues a claim fee balance for a supported `ilk` with an `issuance` and `maturity` to the specified address. No funds are collected. The claim fee balance is supposed to be distributed/sold through various methods to vault owners. The amount issued must be in an 18 decimal representation.

- `withdraw`
    - Allows to burn claim fee balance of any user.
- `initializeIlk`
    - enable support for an `ilk`. The governance is trusted to add only normal `ilks` without special behavior which may not be supported by claim fee.

Unprivileged actions for owners of claim fee balances:

- `collect`
    - Allows to collect the accrued stability fee in DAI.
- `rewind`
    - Rewinds issuance of the claim back to a past timestamp. User must provide DAI to cover for the extra yield which can now be withdrawn.
- `slice`
    - Slices one claim balance into two claim balances at a timestamp.
- `merge`
    - Merges two claim balances with contiguous time periods into one claim balance.
- `activate`
    - Activates a claim fee balance whose issuance timestamp does not have a rate value set. Yield earned between issuance and activation becomes uncollectable and is permanently lost. Required as for `collect()` there must be a stored rate at the issuance timestamp.
- `moveClaim`
    - Transfer claim fee balance to another address.

Users can give approvals to act on their behalf within ClaimFee using `hope()`/`nope()`.

For the calculation of the stability fee to be repaid to the holder of the claim fee balance, the system needs data about the evolution of the rate.

- `snapshot` allows anyone to store a snapshot of the current rate of the given `ilk`
- `insert` Privileged action. Within some safeguards, governance can manually add missing values. Governance is trusted to do this correctly, the safeguards do not entirely prevent errors, added rates may not respect the requirement for monotonic increase of rates with time. Should this happen, the system will break.

Usability of the system, redemption of claim fee heavily depends on existing entries for rates at timestamps. Users may be unable to complete their action and collect their DAI if entries for the required timestamps do not exist. Users must be especially careful when executing `slice` and/or transferring (receiving) claim fee balances.

Considerations for VAT shutdown:

Shutdown of the Claim Fee contract may be initiated by a privileged role at any time or by anyone in case the VAT of the DAI stable coin system has been shutdown.

- `close()` initiation of shutdown.
- `calculate()` set the rate for payout. This allows Maker governance to payout the residual value of the unused claim fee balance in DAI depending on the closure timestamp and the maturity.
- `cashClaim()` allows user to cash out. User needs to provide a claim fee with issuance of `latestRateTimestamp[ilk]` and the respective maturity for wich a ratio has been set by the governance. `Slice()` might be used to split the claim fee into the required parts. The later part starting at `latestRateTimestamp[ilk]` until maturity for `cashClaim()`. With the first part, the stability fee refund may be collected using `collect()`.

`Collect()` must be used to collect the stability fee up to the closure.

Transfer of DAI is done within the VAT accounting only, no actual DAI tokens are transferred.

Considerations for liquidations:

As this addition is separate from the core system, no special considerations must be taken for liquidations. There is no connection between the debt of an urn and the claim fee balance covering this debt. The urn may be liquidated if it reached an unhealthy state. Up to the maturity the claim fee balance can still be redeemed for the stability fee this debt would have accrued.

2. Gate1

Claim Fee generates the DAI required for payout by minting unbacked stablecoin for the VOW. Accessing `vat.suck()` directly would require the contract to bear the `ward` role inside the core Maker system. To mitigate this risk, a gate contract is introduced which will bear the `ward` role in the VAT contract and restrict the access / maximum bad debt that can be generated.

2.3 Trust Model & Roles

Contract ClaimFee:

`wards`: Address with admin authority. Fully trusted to always behave honestly and correctly.

`bud` : Less privileged role than `wards` but fully trusted to behave honestly and correctly at all times. The ClaimFee contract is a `bud` in the Gate1 contract.

`users`: normal users, untrusted.

The issuance of the claim fee balance does not collect any funds. Moreover, compensating the stability fee is done by minting unbacked stablecoin accounted to the VOW. We assume that the claim fee balances issued by the privileged role (the governance) are sold and the collected amount of DAI is forwarded to the VOW. Furthermore, to settle the debt, `vow.heal()` **has to be executed accordingly. The system makes a surplus if the fixed rate exceeds the reimbursed variable rate. Otherwise, the VOW incurs a loss.**

We assume that `jug.drip()` is executed frequently for all supported ilks and the resulting increments in the rate are small.

Furthermore, it is assumed that either snapshots are executed for all ilks regularly, or the governance updates the respective values.

Contract Gate1:

`wards`: Assumed to be the Maker governance exclusively. Fully trusted to always behave honestly and correctly. Most importantly they can set the `approvedtotal` which limits the amount this contract can draw as unbacked stablecoin from the VAT.

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
- 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	1

- Redemption Blocked When No Rate Entry at Maturity Exists Risk Accepted

Low-Severity Findings	3

- Gate1 Withdraw Timestamp Acknowledged
- Leftover Claims Risk Accepted
- Slice at Timestamp With No Rate Risk Accepted

5.1 Redemption Blocked When No Rate Entry at Maturity Exists

Design Medium Version 1 Risk Accepted

After the shutdown of the ClaimFee contract, users may exchange their claim balance for DAI using `cashClaim()`, if it has a maturity after the closure timestamp. However, this requires a valid entry in `ratio[ilk][maturity]` which must be set manually for each `ilk` and `maturity` by the governance.

Since the function `slice` allows users to split their claim fee, many arbitrary maturity timestamps may exist. If the user still holds all segments up to the maturity, they may be able to merge them using function `merge()`. However, these segments may not be available anymore: Individual segments may have been redeemed already, or be unavailable to the user as they have been transferred using the function `moveClaim`.

Overall, users may be blocked and unable to redeem their claim fee.

Risk accepted:

Deco accepts the risk that rate entries might be missing for maturity timestamps. They pledge to provide appropriate support to ensure all maturities have a valid ratio set in case of emergency shutdown.

5.2 Gate1 Withdraw Timestamp

Design Low Version 1 Acknowledged

In the *Gate1* constructor, the `withdrawAfter` timestamp is set. The only check made using this value is to see if it is in the past. Thus, not setting the value at all would save gas and yield the same results.

Acknowledged:

The additional storage write is a one-time cost during deployment.

5.3 Leftover Claims

Design | Low | Version 1 | Risk Accepted

`ClaimFee.collect()` reimburses the stability fee accrued between the issuance and collect timestamp. If the collect timestamp is not equal to the maturity, a new claim fee is issued from the collect timestamp to the maturity.

In general, it's very unlikely that a valid rate is stored for the maturity timestamp: Apart from values manually inserted by the governance, rates stored through function `snapshot()` can only exist for valid block timestamps. The maturity of a claim fee could have been set months in advance upon issuance or the claim fee could have been sliced in various ways. Hence, most of the time, it's not possible to collect up to the maturity timestamp. This design will result in minting many small "leftover" claims.

Risk accepted:

There will be standardized maturity timestamps, e.g.the first day of the month at 12:00:00 UTC. Additionally, it is planned to run bots that regularly take snapshots to ensure that any leftover claims are sufficiently small to be negligible. Lastly, users will be warned against using functionality which creates non-standard maturity timestamps.

5.4 Slice at Timestamp With No Rate

Design | Low | Version 1 | Risk Accepted

The function `slice` allows users to split their claim at a certain timestamp. However, it is possible that the timestamp at which they split their claim does not have a valid rate. Unless they later merge their claims again, or the governance adds a valid rate for the split timestamp, it may not be possible for the user to redeem the full value of the claims.

Risk accepted:

As stated previously, it is intended to have standardized maturity timestamps so that users can know in advance which timestamps will have valid rates. Using such timestamps, users are able to split their claims without incurring any losses. Should the need arise there are two pathways to mitigate the situation: Governance may insert snapshots at timestamp or users can use `activate()` to activate a claim fee balance at a timestamp with a rate set. Note that yield earned between issuance and the activation timestamp becomes uncollectable and is permanently lost.

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	1

- Increasing VAT Debt After Shutdown, After thaw() Code Corrected

Medium-Severity Findings	4

- Comments Regarding vow.heal() Specification Changed
- Governance Can Burn From Users Specification Changed Risk Accepted
- Gate1.heal() Code Corrected
- totalSupply Mapping Not Updated Code Corrected

Low-Severity Findings	6

- Address of VOW Code Corrected
- Duplicate Check Code Corrected
- Maturity in the Past Code Corrected
- Unused Constants and Function Code Corrected
- Various Event Issues Code Corrected Acknowledged
- this Keyword in initializeIlk Code Corrected

6.1 Increasing VAT Debt After Shutdown, After `thaw()`

Design High Version 1 Code Corrected

ClaimFee generates the required DAI by calling `vat.suck()` through the Gate1 contract which acts as a safeguard to enforce a limit on the maximum amount of DAI that can be generated by adding bad debt to the system.

`vat.suck()` is independent of the system status, notably whether the VAT is live or not. Hence, the call to `vat.suck()` will add more bad debt and generate DAI when the VAT is in shutdown. This occurs even after `end.thaw()` has been called in step 6 of the shutdown, which fixes the total outstanding supply of DAI.

The Gate1 contract's purpose is to limit access of the ClaimFee contract in the core maker system: In order to draw bad debt using `vat.suck()` one needs to be a `ward` in the VAT to be able to pass the `auth` modifier. To avoid giving full privileges to the external ClaimFee contract, an intermediary contract Gate1 is introduced, which will be given the privileged role in the VAT. The code of the Gate1 contract enforces limitations in order to limit the risk for the core system. In its current state, the Gate1 contract is missing restrictions to prevent drawing more debt when the VAT is in shutdown.

For further reference:

https://github.com/makerdao/dss/blob/master/src/end.sol#L410
https://docs.makerdao.com/smart-contract-modules/shutdown/end-detailed-documentation#6.-thaw
https://github.com/makerdao/dss/blob/master/src/vat.sol#L230

Code corrected:

A check for the condition `VatAbstract(vat).live() == 1` was added to the `accessSuck` function in the ClaimFee contract. This prevents the debt from increasing after the VAT is in shutdown.

6.2 Comments Regarding `vow.heal()`

Correctness Medium Version 1 Specification Changed

One of the annotations of the Gate1 contract reads:

> • does not execute vow.heal to ensure the dai draw amount from vat.suck is lower than the surplus buffer currently held in vow

There is the following comment in `Gate1.accessSuck()`:

```
// call suck to transfer dai from vat to this gate contract
try VatAbstract(vat).suck(address(vow), address(this), amount_) {
    // optional: can call vow.heal(amount_) here to ensure
    // surplus buffer has sufficient dai balance

    // accessSuck success- successful vat.suck execution for requested amount
    return true;
} catch {
```

- `vow.heal()` uses surplus DAI of the VOW (= surplus buffer) to repay bad debt of the VOW at the VAT
- `vat.suck()` generates DAI by creating bad debt assigned to the VOW

`Vat.suck()` simply adds bad debt, there is nothing ensuring the amount of DAI drawn is lower than the surplus buffer.

Specification changed:

The annotation and comments were removed.

6.3 Governance Can Burn From Users

Correctness Medium Version 1 Specification Changed Risk Accepted

The function `ClaimFee.withdraw()` allows the privileged ward role (the governance) to burn a claim of any user. However, the function's annotation contradicts this as it states the following:

```
/// Withdraws claim balance held by governance before maturity
/// @dev Governance is allowed to burn the balance it owns
```

Furthermore, this function can also withdraw/burn a claim balance upon/after maturity.

Risk accepted:

The annotation was changed to reflect the functionality. The risk of allowing the governance to burn any user's balance is accepted, as they plan to add additional contracts with functionalities that require burning claim fee balances.

6.4 Gate1.heal()

Design | Medium | Version 1 | Code Corrected

`Gate1.heal()` is annotated with:

```
// Access to vat.heal() can be used appropriately by an integration
```

It simply calls `vat.heal()`:

```
function heal(uint rad) external {
    VatAbstract(vat).heal(rad);
}
```

`Vat.heal()` heals bad debt of `msg.sender()`

```
function heal(uint rad) external {
    address u = msg.sender;
    sin[u] = sub(sin[u], rad);
    dai[u] = sub(dai[u], rad);
    vice   = sub(vice,   rad);
    debt   = sub(debt,   rad);
}
```

- The Gate1 contract however doesn't accrue bad debt when generating DAI: Gate1 only draws bad debt using `vat.suck(address(vow), address(this), amount_)`. The bad debt is assigned to the VOW, only the generated DAI is assigned to the Gate1 contract:

```
function suck(address u, address v, uint rad) external auth {
    sin[u] = add(sin[u], rad);
    dai[v] = add(dai[v], rad);
    vice   = add(vice,   rad);
    debt   = add(debt,   rad);
}
```

If the Gate1 contract doesn't accrue bad debt outside of its own functionality, the function has no purpose.

Furthermore, if Gate1 does indeed accrue bad debt, the intended backup DAI balance may be compromised by the fact that anyone could call `heal()` and use some of this DAI balance to heal the bad debt.

Code corrected:

The `heal()` function of the Gate1 contract was removed.

6.5 `totalSupply` Mapping Not Updated

Correctness | Medium | Version 1 | Code Corrected

The *ClaimFee* contract has a `totalSupply` mapping which should track the total supply of claims per ilk. However, neither the `mintClaim` nor `burnClaim` functions update the mapping.

Code corrected:

The `totalSupply` mapping is now updated accordingly in the `mintClaim` and `burnClaim` functions.

6.6 Address of VOW

Design | Low | Version 1 | Code Corrected

In the Gate1 Contract, both the VOW and the VAT addresses are stored as immutables. In contrast, the ClaimFee contract stores both addresses in storage without implementing functionality to update the address.

As reading from storage is expensive, variables set only during deployment may be changed to immutables. During the deployment, all immutable values are inserted into the bytecode of the deployed contract code. Hence, they can be accessed during execution without the need for an expensive SLOAD operation.

Note that there are ongoing discussions to change the VOW to use a proxy: https://github.com/makerdao/dss/pull/241 As such, it may be necessary to have a mutable storage variable for its address.

Code corrected:

The VAT address was made immutable in the ClaimFee contract, and the VOW address was removed. Instead, the VOW address is dynamically queried from the Gate1 contract when necessary.

6.7 Duplicate Check

Design | Low | Version 1 | Code Corrected

The function `issue` checks the following condition:

```
require(initializedIlks[ilk] == true, "ilk/not-initialized");
```

However, the `mintClaim` function checks the very same condition and hence the check in `issue` is unnecessary.

Code corrected:

The duplicate check was removed.

6.8 Maturity in the Past

Design Low Version 1 Code Corrected

The function `issue` places the following requirement on the maturity timestamp:

```
require(
        issuance <= latestRateTimestamp[ilk] && latestRateTimestamp[ilk] <= maturity,
        "timestamp/invalid"
    );
```

However, there is no guarantee that the value `latestRateTimestamp[ilk]` is recent. As it makes little sense to issue a claim with a maturity in the past, one could instead check that the `maturity` is later than the current block timestamp.

Code corrected:

The `issue` function now ensures the condition `block.timestamp <= maturity`.

6.9 Unused Constants and Function

Design Low Version 1 Code Corrected

There are a few constants and a function that are unused or could otherwise be omitted.

1. The `MAX_UINT` constant could be replaced with the built-in Solidity constant: `type(uint256).max`.
2. The constant `RAD` is never used.
3. The function `wmul` is never used.

Code corrected:

The `MAX_UINT` constant was replaced as suggested; `RAD` and `wmul` were removed.

6.10 Various Event Issues

Design Low Version 1 Code Corrected Acknowledged

There are a few functions in which events should be emitted or the event parameters should be indexed.

1. In the *ClaimFee* constructor, no `Rely` event is emitted when the message sender is added as a ward.
2. No event is emitted by the `close` function. This is an important change regarding the functionality of the contract and hence should emit an event.
3. No event is emitted by the `calculate` function. Again, this is an important storage change which allows users to cash out. Indexing the events would allow users to search for specific ilks and maturities.
4. The *Kiss* and *Diss* events in the *Gate1* contract are not indexed.
5. The *NewApprovedTotal* and *Draw* events in the *Gate1* contract could have indexed amounts.

Code corrected:

1. A *Rely* event emission was added to the *ClaimFee* constructor.
2. A *Closed* event was added and is now emitted by the `close` function.
3. A *NewRatio* event was added and is now emitted by the `calculate` function.
4. The address parameter in the *Kiss* and *Diss* events in the *Gate1* contract are now indexed.

Acknowledged:

5. The *NewApprovedTotal* event was removed. The *Draw* event now has an indexed `accessSuckStatus` parameter, but the `amount` parameter is not indexed as Deco did not see the need for it.

6.11 `this` Keyword in `initializeIlk`

Design Low Version 1 Code Corrected

The `initializeIlk` function makes the following call to the `snapshot` function:

```
function initializeIlk(bytes32 ilk) public auth {
    // ...
    this.snapshot(ilk); // take a snapshot
}
```

Calling a function in this way incurs an extra cross-contract call. In order to make an internal call, the snapshot function would have to be declared `public` instead of `external` and the `this` keyword removed.

Code corrected:

In Version 3 the snapshot functions visibility has been changed to `public`, the call in `initializeIlk()` has been updated accordingly.

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 Circumvent `withdrawAfter` Restriction

Note Version 1 Code Partially Corrected

Gate1 features a restriction for function `withdrawDai()`. The privileged role able to pass the `auth` modifier can only call `withdrawDai()` successfully when the withdrawal condition is satisfied:

```
bool withdrawalAllowed = (block.timestamp >= withdrawAfter);
```

The privileged role able to pass the `auth` modifier can always add any address as a `bud` using the function `kiss()`. Such an account can then pass the `toll` modifier and successfully call `suck()` / `draw()` and draw DAI. If the call to `vat.suck()` is unsuccessful (e.g. if the limit has already been reached) this allows to withdraw the backup DAI balance of the contract.

Code partially corrected:

While it is no longer possible to add a `bud` when the withdrawal condition is not satisfied, an already existing `bud` would still be able to circumvent the restriction. For example, after the contract is created, a `bud` could be added, and only then `withdrawAfter` would be set to a timestamp in the future. Alternatively, one could wait for `withdrawAfter` to be in the past, then add a `bud` and set `withdrawAfter` to a future timestamp. Therefore, a `ward` is still able to withdraw the backup DAI balance of the contract.

7.2 Discrepancy Between Reimbursed Amount and Actual Stability Fee

Note Version 1 Risk Accepted

The stability fee paid in in the Maker system is based on the rate increase between when taking and repaying the debt.

ClaimFee reimburses the stability fee based on stored snapshots of the rate.

There are corner cases where the rate stored may not match the actual rate debt was taken/repaid for at this timestamp and hence the reimbursed amount of DAI is not the amount of stability fee paid by the user.

Storing the current rate in ClaimFee does not trigger the update of the rate in the Maker system (`jug.drip()`). A later transaction in the very same block may trigger `jug.drip()` and further transactions modifiying a debt position of this ilk use the new rate.

Consider the following scenarios which must happen within the same block:

1.
   - `ClaimFee.snapshot()` is executed and rate A is stored
   - `Jug.drip()` is executed -> The rate is udpated to A+x
   - The user repays debt in the Maker system at rate A+x

When the user calls `collect()` on his claim fee balance he is reimbursed based on the "old" rate stored and receives less DAI than actual stability fee paid.

2.
   - User takes debt in the Maker system at rate A
   - `Jug.drip()` is executed -> rate is udpated to A+x
   - `ClaimFee.snapshot()` is executed and rate A+x is stored

   Similarly, the user may not be compensated for the full stability fee in this scenario. Note that normally, with the stability fee based on the rate/time, the user has an incentive to increase the rate first using `jug.drip()` before taking on debt. However, unaware users with the impression that claim fee covers their stability fee may not do this.

We assume that `jug.drip()` is executed frequently and the resulting rate increase is small enough so the discrepancies arising in scenarios as described above can be neglected.

Risk accepted:

The risk is accepted based on the assumption that the rate increases are small enough to be negligible.

7.3 No Connection Between ClaimFee and Actual Debt

Note Version 1

There is no connection between an issued claim fee and debt in the VAT. ClaimFee reimburses the stability fee its amount (`art`) would have accrued.

Note that the amount of claim fees issued per `ilk` should not exceed the amount of actual debt per `ilk` otherwise more stability fee is reimbursed than is actually accrued by the system.

Deco responded:

– Our goal is to help the Maker protocol find users who want to hold a vault open for the entire term of the claim fee so that the protocol can derive the benefits of a sticky user and collect the fixed-rate revenue upfront without having to make any re-imbursements later to these users from the revenue generated by variable-rate vaults held by others. We want to ensure claim fee supply stays matched to the vaults who signed up for fixed-rate debt at the issuance date.

– ClaimFee has a transfer function which already allows a vault owner who has claim fee balance and no longer wants to use it to transfer it to another regular vault owner. This would keep claim fee balance less than ilk debt and not trigger the excess reimbursement issue.

– We originally planned to avoid reimbursements that exceed stability fee accrual to the system when debt level drops directly at the urn that was supposed to use the claim fee balance, by combining both the urn and claim fee balance and routing all its usage through a CDP Manager style contract which can create and manage a fixed-rate vault. This CDP Manager can have the required state transitions to keep both debt and claim fee balance of the vault in sync over its lifetime.

– We now plan to design and deploy a much simpler and standalone “Liquidation Penalty” contract instead of the modified CDP Manager. Liquidity Penalty contract can withdraw an amount of claim-fee balance (burn it) held by a user address to match any reduction in debt on a regular vault the same address holds. We don’t want addresses holding claim fee balances standalone without also holding vaults of the collateral type between the issuance and maturity timestamps of the claim fee balance. This liquidation penalty contract could re-imburse the claim fee balance after taking a haircut on its current

value(let’s say 75%, make it attractive for fixed-rate vault owners to abandon their claim fee balance, but not set it too high at like 100% to ensure no reimbursement but force claim fee holders to find buyers among other vault owners to avoid loss of value) to ensure claim fee in circulation stays below the debt held in urn at all times, thereby also solving the issue at the ilk level.