PUBLIC

Code Assessment

of the Direct Deposit V2 Smart Contracts

October 31, 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 Direct Deposit V2 according to Scope to support you in forming an opinion on their security risks.

Direct Deposit Module V2 is a modular framework which allows to generate and deposit DAI into third party systems in order to earn yield. For each supported third party a Plan contract implements the calculations to reach the target state while a pool contract manages the interaction between the D3MHub and the protocol.

The most critical subjects covered in our audit is the functional correctness, security of assets managed and impact/added risk on the existing Maker system.

This iteration of the review focussed on the redesigned implementation of the D3MHub and fixes of issues raised in the last review. The documentation available only gives a high level description of the system, description of detailed behavior (e.g. temporary exceeding debt limits during a transaction) or limitations (unsupported/broken distribution of pool shares in case of loss) is missing.

In summary,apart from the raised concerns when a third party system makes a loss and the pool shares held no longer cover the expected DAI amount, we find that the codebase provides a high level of functional correctness and 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	1
• Code Corrected	1
Low-Severity Findings	11
• Code Corrected	5
• Code Partially Corrected	1
• Risk Accepted	1
• Acknowledged	4

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 Direct Deposit V2 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	12 April 2022	22aaa7af2e5d91d181aac82c595e8c7e9bcb5481	CompoundDAIPlan review only
2	3 May 2022	fa793a3f97bd235cab5e3cd847ed5812735d1557	After Intermediate Report
3	8 July 2022	d5a31c84c43f90e54a4456e291da8b6be74d3626	Start full review
4	6 September 2022	fa9b95d7530bfa18df23849d0f302cf2b32a90f5	New D3MHub implementation
5	4 October 2022	64dac3a3559b358ebdd4ea302a7c7477a8ef1a55	New D3MHub implementation with fixes

For Version 1 and 2:

For the solidity smart contracts, the compiler version `0.6.12` was chosen. The contracts D3MPlanBase and D3MCompoundDaiPlan are in scope.

After Version 3 the review of the full D3M V2 codebase started. This includes the following files:

- D3MHub.sol
- D3MMom.sol
- D3MOracle.sol
- plans/*
- pools/*

For the solidity smart contracts, the compiler version `0.8.14` was chosen.

2.1.1 Excluded from scope

All contracts not listed above and the tests in /tests.

2.2 System Overview

This system overview describes Version 4 of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

The Direct Deposit v2 Module (D3M) is a set of smart contracts that implement the allocation of DAI, according to a plan, to other smart contracts such that MakerDAO generates fees. The architecture is built up modularly around a core such that more integrations to external systems can be plugged in. Initially, the D3M supports managing funds on AAVE (aDAI) and Compound (cDAI).

The core contract of the D3M system is the D3MHub that interacts with the VAT and allocates funds to the modules for the integrations.

Each such module consists of

- a D3M Pool contract that integrates with the external system and holds the funds (LP tokens / Pool Shares),
- a D3M Plan contract that computes the DAI amounts that should be deposited or withdrawn,
- and a D3MOracle which reports a constant price for the special D3M ilks.

To bypass governance delays in emergency situations, trusted parties can route disabling actions through the D3MMom.

2.2.1 D3MHub

The heart of the D3M is the D3MHub, from where each integration specific action are performed using the correct modules. Each such module will be regarded as a special ilk with its gem being the pool shares from the third party system.

As with other ilks, each ilk in the D3MHub is identified by a `bytes32` string and has the following properties:

- a D3M Pool (see D3M Pool)
- a D3M Plan (see D3M Plan)
- a `uint256` variable `tau` which is the minimum time that needs to pass for a caged ilk to be culled (see below)
- a `uint256` variable `culled` which states whether the ilk has been culled.
- a `uint256` variable `tic` which indicates wether the ilk has been caged (`tic!=0`) and gives the timestamp of when the ilk can be culled.

To activate an ilk for the D3M, the pool and the plan must be set through the `file()` functionality which also allows setting the `tau` for a given ilk or setting the VOW or End contracts for the D3MHub. Note that these are authorized functions and that the standard authorization is implemented through `rely()` and `deny()`.

After activation the, D3M's main entrypoint `exec()` is available which either winds (adds DAI to the pool) or unwinds (removes DAI from the pool) according to the plan (see below for more details).

When winding DAI into a module, backed by the expected pool shares to be received the gem balance is increased by the DAI amount to be generated. This amount of gem is then locked as ink which, due to the 1:1 ratio will correspond to the urns art.

By design the accounting in the VAT remains unaware of any changes in the pool shares value held by the D3MPool contract. Excess value may be taken as profit while loss is not handled.

Generally the collateral balance in the VAT (Gem or Ink) tracks the amount of outstanding DAI expected from the third party system.

If the Shutdown mode is not activated, an ilk (in the D3M module) can be shutdown with function `cage()` which sets `tic` according to `tau` and the current timestamp. Caging an ilk can be done once and cannot be undone. While the module is caged `exec()` can be called but can only unwind.

Only if the Shutdown module is not activated and either `tau` for the ilk has passed since its shutdown and, hence, `tic` has been reached, or an authorized `ward` calls the function, the ilk can be culled through `cull()`. Culling essentially writes off debt by calling `grab()` on the VAT which removes the pool's urn's `ink` (collateral) and `art` (debt), removes the ilk's total debt, converts the `ink` to `gem` for the pool, and sends the loss to the VOW as bad debt by increasing its `sin` and the total system loss `vice`. Note that `culled` is set. While the ilk is culled, `exec()` can still be called and will handle this particular case, as explained below.

In case of VAT shutdown the debt write off done in `cull()` can be reverted. This transfers the debt from the VOW back to the urn. The ilk can be unculled through `uncull()` if and only if the VAT is not live anymore and shutdown has been activated. This enables the End Module to take the collateral and ultimately distribute it to DAI holders. The pool's gem balance is moved to the vault increasing its collateral (ink), debt (art) and the ilk's total debt (Art) by the gem balance. This is achieved by calling first `suck()` and the `grab()`. *culled*` is reset which allows `exec()` to be called also during shutdown under certain conditions.

As previously mentioned, `exec()` is the main entry-point for the D3M to wind or unwind for a given ilk. Several paths can be taken from there:

1. Default path: during normal operations, `D3MHub` is managing the funds to match the D3M plan with the internal `_exec()`. The `_exec()` function has 3 subparts:

   1.1 Accounting for fees collection: the target pool provides the DAI value of the shares the D3M Pool holds as `currentAssets`, if `currentAssets > ink`, then the system has accrued fees. The amount of accrued fees is the minimum value between the generated fees `currentAssets - ink`, a safe amount for later computation SAFEMAX `+ art - ink`, and a limit on the ilk debt ceiling `(ink < lineWad ? (lineWad-ink) : 0) + maxWithdraw`. This amount is added to the pool's vault as locked collateral.

   1.2 Generating new debt if `art < ink`: this happens when the system accrued fees or permissionless DAI was sent to repay the debt. `_exec()` generates a new system debt with the difference of `ink` and `art` to reach the equilibrium `art = ink` with `suck()` and `grab()`.

   1.3 Winding / unwinding: if the ilk is caged, the plan is inactive, or amount of shares returned by the 3rd party is smaller than the `ink`, unwind as much as possible to repay the debt. Otherwise, compute the temporary unwind amount as the maximum across the amount exceeding the ilk's line, the amount exceeding the total system debt, and the amount to remove in order to reach the `targetAssets` from the plan. If the temporary unwind amount is non-zero, the final unwind amount is set as the minimum between the temporary unwind amount (what the system wants to unwind) and the `maxWithdraw` amount (what the system can unwind). If the final unwind amount is non-zero, the amount is withdrawn from the pool and used to repay the debt. If there is no amount to unwind, `_exec()` computes the amount to wind into the pool. The wind amount is computed as the minimum value across the amount to reach the ilk's debt ceiling, the amount to reach the system's debt ceiling, the amount to reach the `targetAssets` from the plan, and the pool's maximum deposit amount. If the wind amount is non-zero, it is used to generate debt and DAI, backed by the pool's shares, and the wind amount of DAI is sent to the pool. Note that the wind and unwind paths are mutually exclusive.

   Note that under certain conditions, `_exec()` will increase the `ink` and `art` of an ilk even if it leaves the total system debt above the total debt ceiling `Line`. This may happen when the `debt` is already above the `Line` or when the `Line` is exceeded during the execution of `_exec()`, and the `maxWithdraw` amount is not sufficient for the `debt` to be reduced down to the `Line`.

2. The ilk has been culled: The system tries to withdraw as much DAI as possible from the pool with `_wipe()`, this DAI is sent to the VOW and the pool's `gem` is reduced by the same amount (or by the maximum gem available). Note that in that scenario, the pool has no debt since its debt has been written off.

3. VAT is not live and emergency shutdown has been initiated: The system debt must not be set yet in the End module and the ilk must be unculled. The target pool's vault will be settled at the tagged price (1:1) and the backing collateral is sent to the End module, this will increase the system's debt by the same amount. This will not add `gap` since the price is constant at 1:1 and `ink>=art`. In order to reduce that debt, `exec()` will now try to `withdraw` as many DAI from the integrated system as possible. This will move the DAI back to the Vow module to help settle the system debt and the backing collateral is removed from the End module vault. Any excess locked collateral will be locked in the D3M pool contract. Note that for unculled pools, that may have been culled in the first place because the pool had an issue, the value of their shares may worth less than 1 DAI each. Users will be able to redeem their gem as a share of a pool's pool token holdings.

Note that the hooks `preDebtChange()` and `postDebtChange()` for the pool will always be executed respectively before and after any action taken by the `exec()` function.

The `exit()` function will reduce a user's gem balance with `slip()` (in the case of D3M ilks it is only possible to have one during shutdown) and calls the `exit()` function on the target pool to distribute the pool shares (LP tokens) held by the D3M pool. Please, see the MCD Caged mode's description above.

2.2.2 D3M Pool

The D3MPool essentially is a wrapper around the integrated system. While the integration relevant parts will differ, the functionality and its expected behaviour can be summarized as follows.

The core function for the D3M pools allow depositing and withdrawing DAI:

- `deposit(uint256 wad)`: Function for the hub to deposit `wad` DAI into the external system.
- `withdraw(uint256 wad)`: Function for the hub to withdraw `wad` DAI from the external system.

Moreover, it implements the following function for the shutdown process:

- `exit(address dst, uint256 wad)`: Function for the hub to transfer the remaining pool shares to `dst` proportionally to the `wad` amount (in GEM). The amount to be sent to `dst` is computed as `(remaining shares) * wad / (ilk's Art - exited)`, where `exited` is the amount that has already been exited. By doing so, DAI holders receive an amount of shares proportional to the DAI they sent to repay the bad debt of the `ilk`, so if the shares-to-DAI ratio loses the 1:1 peg during shutdown every DAI holder is treated equally. Required during shutdown.

A function is offered to the governance for moving the funds:

- `quit(address dst)`: Function for authorized addresses to transfer the full pool share balance held to `dst`. Assumed to be only possible when the system is not in Shutdown. This function can break the collateral backing of the generated ink/art of the pool, as no accounting is done. Authorized addresses are trusted to take this in account and adapt their action flow consequently. (An example of such flow can be found here)

As some systems may require some tracking or logic before interaction is done, the following hooks are implemented:

- `preDebtChange()`: Hook before debt changes. Note that currently only the CompoundD3MPlan implements this hook for `exec()` to accrue interest for the cDAI token such that reading state through the variables is accurate.
- `postDebtChange()`: Hook after debt changes. For now, no pool has any logic in the hook.

The following `view` functions are offered (besides integration specific variables):

- `assetBalance()`: Returns the balance of this contract in DAI (derived from pool shares balance).
- `maxDeposit()`: Returns the maximum depositable DAI amount.
- `maxWithdraw()`: Returns the maximum withdrawable DAI amount.
- `redeemable()`: Returns the redeemable token (if it exists).

Furthermore, the standard authorization methods `rely()` and `deny()` are implemented. Note, that `file()` allows setting a new hub and a `king` who will receive extra rewards which can be collected through `collect()`. Not all future integrations are expected to have such a mechanism.

2.2.3 D3M Plan

Each plan implements the on-chain calculations of the target supply for the integrated with pool. Its essential function is `getTargetAssets()` which, given the current assets held by the pool as input, reports the DAI supply that should be targeted such that the stored value `barb` (target interest rate) is reached.

The computations are typically the inverse of the external system's computation of the interest rate:

- For D3MAavePlan it is the inverse of the DefaultReserveInterestRateStrategy contract's `calculateInterestRates()` function (for the variable borrow interest rate).
- For CompoundD3MPlan is the inverse of the BaseJumpRateModelV2 contract's `getBorrowRate()` function (computation for the variable borrow rate).

Each D3M Plan contract can be queried to ensure that the plan is still valid through the `active()` function. The function will return false if the target interest is `0` or if the 3rd party pool contract modified its parameters.

A plan can be disabled by calling `disable()` if `active()` returns false or if the caller is authorized.

2.2.4 D3MOracle

Per D3M Ilk an oracle will be deployed that will report a constant 1:1 price for the ilk. It implements the following functionality:

- Standard MakerDAO access control management through `rely()` and `deny()`.
- Upgrading the D3MHub through `file()`.
- `peek()`: Returns `10**18` and the validity of the result (VAT is live and ilk is not culled).
- `read()`: Returns `10**18` and reverts if the result is not valid.

2.2.5 D3MMom

The D3MMom is a contract that allows an immediate disabling a pool in the D3M such that governance delay can be bypassed in emergency situations.

It implements the following functions:

- `setOwner()`: Set the owner. Only callable by the governance.
- `setAuthority()`: Set the authority that can bypass the governance delay. Only callable by the governance.
- `disable()`: Callable by either the owner or the governance to disable a D3M plan which disables the D3M Pool and, hence, only allows withdrawing from the external system.

2.2.6 Trust Model & Roles

Wards: Each address set to 1 in the wards mappings is fully trusted and expected to behave correctly.

Aave: Aave's contracts are central to the functionality of the D3M and are upgradeable. Hence, Aave is fully trusted to not make any malicious modifications to the contracts.

Compound: Compound's contracts are central to the functionality of the D3M and are upgradeable. Thus, Compound is fully trusted not to make any malicious modifications to the contracts.

MakerDAO assumes that both protocols return a positive interest rate such that `assetBalance() >= ink`.

In the case where Aave or Compound (or any external protocol that will be added to the hub later on) face a problem that would trigger an asset loss, it could break the latter assumption. Note that the debt generated for each ilk is only protected by an upward debt ceiling. This means that the loss case is not handled but is only partially mitigated.

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	0
Low-Severity Findings	6

5.1 Aave Pool Size Manipulation

Correctness Low Version 3 Risk Accepted

To calculate the amount to be deposited into the Aave pool to reach the target interest rate is computed in `D3MAavePlan.getTargetAssets*()` by computing the difference of target pool size computed and the current pool size. However, estimation of the current pool size

```
uint256 totalPoolSize = dai.balanceOf(adai) + totalDebt;
```

is prone to manipulation. Note that Aave offers flashloans that do not update the debt. Hence, it is possible to manipulate the pool size and hence the amount moved into the pool by first flashloaning on Aave and then calling `exec()`. nac

---

Risk accepted:

MakerDAO responded:

```
This is a concern we discussed at-length during our internal review. We have
identified several scenarios where the D3M could be manipulated in this way.

The high level conclusion we came to in our internal review was that such a
manipulation would 1) likely not have a high impact to the system and 2) would
be relatively short lived.
```

5.2 Disable Plan Without Event

Design | Low | Version 3 | Acknowledged

The `disable` function of both, D3MAavePlan and D3MCompoundPlan sets the target interest rate to 0 and emit the Disable event.

Both contracts feature a `file` function which allows to set the target interest rate to 0 without the Disable event being emitted.

Acknowledged:

The “Disable” event is emitted when the contract is disabled using the `disable` function. This function can be called permissionlessly if the plan leaves the active state.

When the Maker governance uses the file function to set `bar` to 0, the "File" event is emitted. This pattern matches the pattern elsewhere in Maker contracts when parameters are changed by authorized users.

Analysts monitoring the contract for shutdown will need to look for both *Disable()* and *File(“bar”, 0)* in their event parsing scripts.

5.3 Inconsistencies

Design | Low | Version 3 | Code Partially Corrected | Acknowledged

Similar contracts differ at similar places but could be more consistent. For example:

- The D3MAavePool does not validate that the aDAI address is non-zero while the D3MAavePlan does.
- `file()` for D3MCompoundPlan validates the target interest rate against the maximum while the D3MAavePlan treats it as special case.
- The D3MCompoundPool performs a == check after `deposit()` while the D3MAavePool performs a >= check.
- The target interest rate is `barb` in the D3MCompoundPlan and `bar` in the D3MAavePlan.

Pre- and post-conditions may be treated as documentation and, hence, having them consistent and similar may clarify the assumptions the D3M Hub makes about the modules' behaviour.

Code corrected:

- The D3MAavePool now validates that the aDAI address is non-zero.

Acknowledged:

- An additional check in `file()` is unnessesary as this must be validated within `D3MAavePlan._calculateTargetSupply()`. MakerDAO responded that the additional check in `D3MCompoundPlan.file()` is to prevent spell crafters from shooting themselves in the foot as the block based borrow rate Compound uses does not feel intuitive. Generally the policy is to leave the `file` functions as simple as possible and put guard rails into other contracts such as dss-exec-lib.
- The strict equality is desired, however in D3MAavePool there may be a 1 wei rounding error depending on state hence the looser requirement.

- MakerDAO responded: In this particular case, bar is a per-year interest rate in RAY units, while barb is a per-block interest rate in WAD units. To name these variables the same for consistency would likely cause spell crafters, risk, and governance to make a massive mistake in setting the target borrow rate in the future. For this reason, we named them differently. That is, they are deliberately inconsistent for safety and security reasons.

5.4 `end.skim()` May Leave `ink` Behind

Correctness | Low | Version 3 | Acknowledged

Normally, the `ink` / `art` of the pools urn at the VAT should be at a 1:1 ratio. Anyone however may use `frob()` and by supplying DAI one can reduce any urns debt.

Through the code of D3MHub `_fix()` is used to fix the urn. In one corner case this is not done:

Just before the VAT is caged, someone repays debt of the urn of a pool. `art` is now less than `ink`. After the VAT is caged `exec()` is called. The following code is executed:

```
} else if (mode == Mode.MCD_CAGED) {
// MCD caged
// debt is obtained from free collateral owned by the End module
_end = end;
_end.skim(ilk, address(_pool));
daiDebt = vat.gem(ilk, address(_end));
```

`end.skim()` settles the debt of the urn by confiscating `ink`. As not all `ink` is needed to cover the `art` of the urn, there is some `ink` remaining. This collateral will be locked forever.

Acknowledged:

MakerDAO states:

```
This is an acceptable edge case that we assess to pose little or no risk. The
actor that “donates” DAI to create the situation loses value, and the effects
on other actors are minimal. DAI holders receive the same distribution they
would have had the donation not occurred (although in principle they could have
received even more had the donation been taken into account). Vault holders are
not affected. The external lending market now technically has a certain amount of
locked DAI lending supply, but given the fundamental shift in the nature of DAI due
to, Emergency Shutdown of the Maker protocol, this is unlikely to matter at all.
While special-case logic could be added in `exec` to account for this, it likely
isn’t worth the extra complexity.
```

5.5 Immutable InterestRateModel

Correctness | Low | Version 1 | Acknowledged

In the D3MCompoundDaiPlan contract, InterestRateModel is marked as immutable :

```
InterestRateModel public immutable rateModel;
```

This field corresponds to the InterestRateModel field of the CErc20 money market that has DAI as underlying asset. The CErc20 contract inherits from the CToken contract, which means the InterestRateModel can be updated :

```
function _setInterestRateModel(InterestRateModel newInterestRateModel) public returns (uint) {
    /*...*/
}
```

An update of the InterestRateModel field of the CErc20 contract cannot be reflected in the D3MCompoundDaiPlan contract since the InterestRateModel field is marked as immutable.

The stored `rateModel` is used in function `_calculateTargetSupply`. Should the interest rate model of the cDAI token be updated unexpectedly, the calculations may be incorrect.

Acknowledged:

Maker acknowledges the issue and states:

```
If the rate model changes unwinding can be permissionlessly triggered through the hub's `cage` function.
We are now also working on having that block `exec` immediately (on a separate branch).
```

Update: As of Version 3, if the interest rate model changes, `exec` will trigger an unwind as though the pool is caged.

5.6 Optimization of `auth` Modifier

Design | Low | Version 1 | Acknowledged

The `auth` modifier of the D3MPlanBase contract checks the condition `wards[msg.sender] == 1`. However, as the ward mapping only contains values of 0 and 1, it is sufficient to check that `wards[msg.sender] != 0`. This results in a slightly more efficient compilation of the modifier.

Acknowledged:

Maker prefers to stay with the current implementation as it is used in other Maker repos.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Unclaimable or Leftover PoolShares During Global Settlement Code Corrected

Low-Severity Findings	5

- Unwind Collects Interest, Fails to Reach Target Interest Rate Code Corrected
- Discrepancy in the Handling of Unachievable Target Interest Rates Code Corrected
- Insufficient Conditions for active() Plan Code Corrected
- No Events Code Corrected
- No Natspec Code Corrected

6.1 Unclaimable or Leftover PoolShares During Global Settlement

Correctness Medium Version 3 Code Corrected

When winding DAI into a module, backed by the expected pool shares to be received, the `gem` balance is increased by the DAI amount to be generated. This amount of `gem` is then locked as `ink` which, due to the 1:1 ratio will correspond to the urns `art`.

By design, the accounting in the VAT remains unaware of any changes in the pool shares value held by the D3MPool contract. In normal operation, surplus is handled by taking the profit while the loss case is generally unhandled. Please refer to the corresponding open question at the end of this report.

If the VAT is caged, calling `end.skim()` cancels all of the owed DAI from the Vault and assigns the freed `ink` collateral to the END which is later distributed amongst all DAI holders.

The value of the PoolShares held by the D3MPool contract however may changes: For rebasing tokens, the amount of tokens may change (Aave aDAI). For others the exchange rate may change (Compound cDAI).

In the late state of the shutdown users will receive of these D3M `gems`. They can then redeem these `gems` for the underlying using `D3MHub.exit()`.

In case of `aDAI` the call to `D3MAavePool.transfer()` will transfer the amount of `gem` in aDAI.

In case of `cDAI` the call to `D3MCompoundPool.transfer()` will transfer the amount of `gem` divided by `cDai.exchangeRateCurrent()`, which translates to the current amount of cDAI token which corresponds to this DAI amount.

The third party systems are independent of the VAT. These may continue to operate normally and accrue more interests during the Shutdown process. Or they may be excess profit which has not been collected yet.

Leftover collateral tokens will remain at the D3MPool contracts:

- D3MAavePool: The aDAI balance held exceeds the sum of `gem`, hence not all aDAI can be distributed.
- D3MCompoundPool: Due to a favorable `cDai.exchangeRateCurrent()` not all cDAI will be consumed when users exit their `gem`.

Whenever there was a loss, either before or during the shutdown process, not all `gem` can be redeemed:

- D3MAavePool: The aDAI balance held is insufficient to redeem all the `gem`, hence not all `gem` can be redeemed.
- D3MCompoundPool: The available cDAI balance is insufficient to redeem all `gem` with the current `cDai.exchangeRateCurrent()`, hence not all `gem` can be redeemed.

Code corrected:

The gem balance is now treated as a share of the pool's LP token balance. Hence, users will be able to withdraw their corresponding share.

6.2 Unwind Collects Interest, Fails to Reach Target Interest Rate

Correctness | Low | Version 3 | Code Corrected

`exec()` determines the current state and either winds or unwinds as necessary. If no other constrains apply, it attempts to have supplied `plan.getTargetAsset()` amount of DAI. The implementation of `unwind()` however may remove more assets and the resulting pool state is not exactly as targeted.

This is due to the implementation of `unwind()`. The README states: `Upon unwinding, interest will automatically be collected.`. Interests however are removed in addition to the calculated supply reduction reducing in less than the calculated amount of DAI remaining in the pool.

The implementation of `unwind()` first calculates the amount to unwind based on the calculated `supplyReduction` and constraints:

```
// Unwind amount is limited by how much:
// - max reduction desired
// - assets available
// - dai debt tracked in vat (CDP or free)
uint256 amount = _min(
                    _min(
                       supplyReduction,
                       availableAssets
                    ),
                    daiDebt
                );
require(amount <= MAXINT256, "D3MHub/overflow");
```

and later adds the fee on top for the amount to withdraw.

```
// To save gas you can bring the fees back with the unwind
uint256 total = amount + fees;
```

```
//uint total = amount;
_pool.withdraw(total);
```

As too many DAI have been removed, the utilization is higher and the target interest rate is not reached. A second call to `exec()` could rectify this by resupplying the missing amount of DAI.

Code corrected:

The implementation of `exec()` has been redesigned, the issue described above no longer exists.

6.3 Discrepancy in the Handling of Unachievable Target Interest Rates

Correctness | Low | Version 1 | Code Corrected

There is no upper bound on the value of `targetInterestRate` in the `_calculateTargetSupply` function, nor on `barb` in the `file` function. Consequently, it is possible to obtain a target utilization rate (`targetUtil`) above 100% in `_calculateTargetSupply`. This is not possible in Compound, thus the target interest rate is unachievable. However, `_calculateTargetSupply` will return a non zero value as if the target rate was achievable.

This is not consistent with the behavior of `_calculateTargetSupply` in the other cases where the target interest rate is not achievable: When `targetInterestRate` is above `normalRate` but `jumpMultiplierPerBlock` is zero, or when `targetInterestRate` is below `baseRatePerBlock`, `_calculateTargetSupply` returns 0.

Code corrected:

`_calculateTargetSupply` now returns 0 when the calculated utilization is over 100%.

6.4 Insufficient Conditions for `active()` Plan

Correctness | Low | Version 1 | Code Corrected

Plans can be manually disabled or enabled. However, that may also happen automatically. For example, the D3MCompoundPlan will become inactive if the implementation contract has changed. While the checks for both plans are rather extensive, there could be some properties that could also be considered for the `active()` `view` function.

For example:

- Aave does not offer querying the implementation contract such as compound but offers `getRevision()` which returns the version of the AToken. Similarly, that holds for other Aave contracts such as the lending pool.
- Before depositing into Aave, Aave does a `validateDeposit()` check which checks if an AToken is active and not frozen.
- Before depositing into Compound, the CToken calls `mintAllowed()` on the Comptroller to check if the market is listed or paused.

Were these and similar properties considered and why aren't they use in the `active()` function?

Code corrected:

MakerDAO responded:

```
The first check for Aave (ATOKEN_REVISION) is very helpful and has been added.

The other two concern a paused or inactive token in Aave or Compound. We explored
this during our internal review and unfortunately, found that this is not a helpful
check to add to `active`. When a plan returns `false` for active, then we attempt
to unwind as much of our position as possible. In the case where the AToken is not
active/frozen or the CToken is not listed/paused, any transfer of those tokens will
revert so we will not be able to withdraw.
```

6.5 No Events

Design Low Version 1 Code Corrected

Neither function `file` nor `disable` of the D3MCompoundDaiPlan contract emit an event after the parameter `barb` has been changed. Normally `file` functions of Maker projects emit an event.

Code corrected:

Events are now emitted.

6.6 No Natspec

Design Low Version 1 Code Corrected

The external functions, although being view functions only, feature no description / natspec. Is this intentional? Description may allow user to better understand the function parameters (e.g. the `currentAssets` of `getTargetAssets()`) and the return values, thus potentially avoids errors. For example it's not immediately obvious if some parameters are in cDAI or DAI.

Code corrected:

MakerDAO added NatSpec documentation.

7 Open Questions

Here, we list open questions that came up during the assessment and that we would like to clarify to ensure that no important information is missing.

7.1 Checks on `deposit()`/ `withdraw()`

Open Question **Version 1**

Both pool implementations implement sanity checks upon deposit in order to ensure the expected amount of pool shares has been received.

Upon withdrawal the D3MHub enforces that a sufficient amount of DAI is present. Otherwise, the transaction reverts when `DAIJOIN.join()` fails.

Generally, this checks whether redeeming pool shares resulted in the expected amount of DAI, except in corner cases when there was additional DAI balance present for the D3MHub.

A situation where redeeming pool shares results in less than expected DAI is not detected by the D3MHub contract. While the transaction reverts, the D3M hub will remain unaware. A later call to execute, after the utilization of the third party has changed, may wind more DAI into this broken system.

Should there be an on-chain protection mechanism or will this be handled by off-chain monitoring and then disabling a pool through the mom?

8 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.

8.1 Anyone Can Manipulate the Allocation

Note Version 1

By design, the plans' computations on how much to withdraw/deposit heavily depends on on-chain state: The plans aim to reach a target interest rate which is depending on the current utilization ratio. This depends on:

1. the current pool size
2. the currently borrowed amount

Note, that both can be easily manipulated before `exec()` is called. This may be done within the same transaction or in separate transactions. However, this is almost like the normal intended use of the system: Pool states change after interactions, `exec()` is used to return the pool to the desired stated.

The current pool size for example can be manipulated by sandwiching `exec()` as follows:

1. Deposit DAI into the external system's pool by minting LP tokens.
2. `exec()`
3. Withdraw DAI by burning LP tokens.

Note that such manipulation can easily be achieved at low cost to a certain degree since, for example the DAI flash mint module, offers DAI flash loans for free.

The second one can be manipulated by sandwiching `exec()` as follows:

1. Borrow a significant amount of DAI from the pool.
2. `exec()`
3. Payback the loan.

Note that this manipulation requires some extra steps such as depositing collateral to borrow DAI. However, the needed collateral amount could be either already held by a malicious user or could be flash loaned at potentially low cost.

In summary, `exec()` always attempts to reach the target utilization. In a three-step process, first by modifying the state (e.g., depositing DAI minting pool shares), calling `exec()` which now winds/unwinds into the wrong direction to temporarily reach the target utilization. Finally, the attacker may undo the state manipulation of step 1) (e.g., returning the pool shares he borrowed) and the pool utilization is significantly off.

As long as `exec()` works as intended, this is no issue. A subsequent call to exec can return the pool to the desired state. Note that in corner cases this may not be possible: Limitation from `maxDeposit()` / `maxWithdraw()`, `Line` or available liquidity to withdraw DAI may prevent this. Should an attacker manage to trick `exec()` to wind/unwind but a subsequent `exec()` cannot undo this (temporarily), this is problematic.

A potentially costly attack could be to borrower much DAI over a long time such that the position cannot unwind properly. A variation of this could be front-running calls to `exec()` which would unwind and remove the available DAI liquidity.

8.2 Caging Arbitrary Ilks

Note **Version 1**

Technically, for the governance it is possible to cage ilks to either do not exist yet or that are not D3M ilks. Potential consequences could be:

- The non-existing ilk is added to the system and could be immediately culled.
- The non-D3M ilks could be added that could be similarly culled. However, if `0x0` has some `ink` or `art` for that ilk, the `ink` will be converted to `gem` while the debt will be written off.

8.3 Document VAT Shutdown

Note **Version 1**

VAT Shutdown considerations are not documented for this special ilk type. Due to the special nature / behavior such documentation should be readily available and may include:

- Description of the different states a D3M ilk could be in
- Effects of unwinding during shutdown (`MCD_cage` mode) and a description of the process
- Culling and what implications a culled ilk during shutdown has
- Unculling and the reasons when it could be worth calling it, and the conditions for when `uncull()` can be called
- Considerations when ilks were not fully unwinded (e.g. D3M oracles can not be queried and hence the ilk cannot be caged)

8.4 Exposure to New Collaterals / Markets

Note **Version 1**

Currently the set of `ilks` / collaterals backing the DAI Stablecoin is rather restricted to well known and trusted assets only.

D3MHub lends DAI to third party protocols such as AAVE and Compound. This results in exposure to new markets and collateral assets. The risk can be limited through the `Line` / debt ceiling set for the corresponding `ilk`.

8.5 Maximizing Revenue

Note **Version 1**

By supplying DAI into lending protocols such as Compound/Aave the protocols utilization is reduced and borrowing DAI gets cheaper.

With DAI generated through D3M Maker only profits from interests accrued by the DAI in the third party protocol in contrast to DAI generated with normal ilks where users have to pay the stability fee.

With Compound/Aave users earn interest on their collateral while paying interest for borrowed DAI. The users position has to be overcollateralized, hence the supply interest are earned on a larger amount compared to the borrow interest on the smaller DAI amount.

There is a risk that users can get DAI cheaper via these protocol compared to using the Maker Dai Stablecoin system if the parameters for the target borrow rate / pool utilization are not chosen carefully.

E.g. at the time of writing (July 17th, 2022):

Aave V2: The variable borrow APY for DAI is 1.69% at a pool utilization of 33.46% while the supply APY for Ether is 0.08%. Compared to the ETH-A stability fee in Maker is 2.25%.

Compound: The borrow AP for DAI is 1.79% with a pool utilization of 31.19%. The supply APY for LINK token is 0.43% Compared to the Link-A Stability fee in Maker of 2.50%.

The target interest rate must be chosen carefully taking into account the different stability fees of different `ilks` for the D3M to be worthwhile. Depending on the state of the lending pool (e.g. low utilization / low rates) this may not be possible.

8.6 Unhandled Loss Case

Note **Version 1**

When winding DAI into a module, backed by the expected pool shares to be received the `gem` balance is increased by the DAI amount to be generated. This amount of `gem` is then locked as `ink` which, due to the 1:1 ratio will correspond to the urns `art`.

By design the accounting in the VAT remains unaware of any changes in the pool shares value held by the D3MPool contract.

If the module generates a profit in form of interests the profit is accounted for.

If the third party system makes a loss, the value of the pool shares held by the D3M pool may decrease. `exec()` does not catch this and may continue to wind DAI into the module if the plan asks to do so. Such a module / ilk must be caged manually by the governance.

MakerDAO replied:

```
Note that it is true that when there is a problem, such as a hack in Compound
or Aave, the exchange rate or rebalancing can be altered both ways.

We think this risk is not significantly different from existing risks in
non-immutable collateral types, and is limited by debt ceilings.
```

8.7 VAT Debt Could Increase After `END.thaw()`

Note **Version 1**

After the processing period of the shutdown, the call to `END.thaw()` will fix the total outstanding supply of DAI according to `VAT.debt()`.

`uncull()` can still be called. This will `suck()` and then `grab()` such that the total debt of the VAT is increased and, thus, it could be possible that the total outstanding DAI supply is increased even after `END.thaw()`.

MakerDAO states:

```
We are aware of the importance/need to uncull all culled D3Ms and will be working
to add this to our End Keeper processes. This is similar to the importance that
all collaterals get skimmed in the waiting period.
```