PUBLIC

Code Assessment

of the DFM Core Smart Contracts

June 13, 2024

Produced for

DEFI.MONEY

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help DeFi.Money with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of the core protocol contracts of defi.money (DFM Core) according to Scope to support you in forming an opinion on their security risks.

DeFi.Money implements a stablecoin system based on Curve's LLAMMA architecture. The changes include architectural changes, aggregation of market data for efficiency and the introduction of hooks.

The most critical subjects in our audit are functional correctness, access control and the correct adaption of the existing Curve code. The general subjects covered are documentation and error handling.

In summary, we find that the codebase provides an good level of security.

Note that the audit focused on the diff with Curve. In case there is an issue in Curve, it might be present in the audited codebase in scope.

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	14
• Code Corrected	13
• 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 assessment was performed on the source code files inside the DFM Core 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	16 May 2024	ce1364ad1142251c042ab31bf643dedfd190ca64	Initial Version
2	1 Jun 2024	b8e7cc37975a7f4c8cfbf466218e2515bc899b02	After Intermediate Report
3	4 Jun 2024	3ee30929aeb3e9a9368b0f12125f31da78c6776d	Further fixes and changes
4	12 Jun 2024	1015505d237d6032c8f63d3763f7b1557d9ee6d1	Fixes

For the Vyper smart contracts, the compiler version `0.3.10` was chosen. For the Solidity smart contracts, the compiler version `0.8.25` was chosen.

The files in scope were:

```
contracts/PegKeeperRegulator.vy
contracts/MarketOperator.vy
contracts/PegKeeper.vy
contracts/AggMonetaryPolicy2.vy
contracts/MainController.vy
contracts/AMM.vy
contracts/interfaces/ICoreOwner.sol
contracts/StableCoin.sol
interfaces/IControllerHooks.sol
interfaces/IPriceOracle.sol
interfaces/IAmmHooks.sol
```

In version 2, the repository has been restructured. Note that the `IAmmHooks.sol` has been removed from the scope and that `ICoreOwner.sol` has been renamed to `IProtocolCore.sol`. The files in scope are:

```
contracts/cdp/PegKeeperRegulator.vy
contracts/cdp/MarketOperator.vy
contracts/cdp/PegKeeper.vy
contracts/cdp/AggMonetaryPolicy2.vy
contracts/cdp/MainController.vy
contracts/cdp/AMM.vy
contracts/bridge/StableCoin.sol
contracts/interfaces/IProtocolCore.sol
contracts/interfaces/IPriceOracle.sol
interfaces/IControllerHooks.sol
```

In version 3, `StableCoin.sol` has been renamed to `BridgeToken.sol` and `IBridgeToken`, `DataStructures.sol` and `MarketViews.vy` was added to the scope. The files in scope are:

```
contracts/bridge/dependencies/DataStructures.sol
contracts/cdp/PegKeeperRegulator.vy
contracts/cdp/MarketOperator.vy
contracts/cdp/PegKeeper.vy
contracts/cdp/AggMonetaryPolicy2.vy
contracts/cdp/MainController.vy
contracts/cdp/AMM.vy
contracts/bridge/BridgeToken.sol
contracts/interfaces/IBridgeToken.sol
contracts/interfaces/IProtocolCore.sol
contracts/interfaces/IPriceOracle.sol
contracts/periphery/views/MarketViews.vy
interfaces/IControllerHooks.sol
```

Given that the audit was an audit of the differences between DFM Core and Curve, the compared to commit was `c08a3ab8eb29d7622eddf432cb518eeec6f88b63` of the `curvefi/curve-stablecoin` Github repository.

2.1.1 Excluded from scope

All files not mentioned above were not in the scope of this review. Further, note that the review focused on the differences introduced by DeFi.Money (compared to the Curve codebase). Any issues present in Curve might be present in DFM Core.

Further, we exclude any external libraries (such as LayerZero OFT implementations) from the scope. We expect the system interacts only with standard ERC-20 (e.g. non-rebasing, no fees). Please see the System Overview and Notes section for more details.

2.2 System Overview

This system overview describes the initially received version (Version 1) 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.

DeFi.Money implements a stablecoin system that builds upon the crvUSD infrastructure. This section will focus on the system parts that have been adapted. See also our crvUSD audit for more details.

2.2.1 Lending

The core of the stablecoin mechanism is the lending module where users can borrow stablecoins. Ultimately, that creates positions on an AMM that allows for soft-liquidations (see also Curve's LLAMMA docs).

2.2.1.1 Core

The main entry point of the system is the `MainController` for users to create, adjust and close loans.

Regular users operate on their position with

1. `create_loan` to create a loan if the user has no loan is active,

2. `adjust_loan` to adjust an existing loan (adds a debt and collateral delta to the loan, however, cannot open or close a loan),
3. and `close_loan` close an existing loan which repays all debt and returns the provided collateral.

Note that users can give addresses approval to call the loan-manipulating functions on their behalf (e.g. router contracts). Addresses are approved with `setDelegateApproval`. The loan manipulation will be performed on the delegator's position but the transfers will be done to and from the delegate.

The loan-adjusting functions will forward market-specific loan-accounting logic to so-called `MarketOperator` contracts. However, the `MainController` will wrap the accounting with other logic such as collateral transfers, stablecoin mints and burns or interest rate updates. Additionally, a global debt ceiling (over all markets) is enforced when new debt is created. Further, hooks are invoked (see Hooks).

The `MarketOperator` keeps track of the individual loans and the market-specific state while integrating with the `AMM` which includes depositing and withdrawing liquidity. The operator additionally implements an additional debt ceiling that is enforced when new debt is created on the market.

Since loans may become unhealthy, two liquidation mechanisms, as in Curve, are implemented. Namely, the soft liquidations are performed with token exchanges. Thus, the AMM implements `exchange` and `exchange_dy` to allow swapping tokens on the AMM. Note that the AMM is not a standard AMM but is designed with the LLAMMA architecture in mind (see links above for more details). In case there is an unhealthy loan where the soft-liquidation mechanism did not succeed in liquidating in time, liquidators can liquidate it with the `MainController.liquidate` function. Similarly, to the loan-manipulating functions, the `liquidate` function wraps the corresponding `MarketOperator` function. Note that the AMM exchanges will also trigger hooks.

`MainController.collect_fees` permissionlessly allows to collect the fees generated over a set of markets. Simply put, fees are the difference between the sum of all stablecoin burns and the outstanding debt and the sum of all stablecoin mints. Note that the corresponding `MarketOperator` of each specified market operator will be invoked to collect the fees generated by the AMM.

Note that fees are also generated by the interest of the lending market. That interest is defined by monetary policies which will be queried after the loans have been adjusted. The interest rates are constant between the interest rate updates (see Monetary Policy).

Governance can adjust the parameters of the contracts mentioned above. More specifically, the following functions are available to governance-only:

1. `MainController.add_market`: Creates minimal proxies for the operator and the AMM that point to the currently set implementations for the `A` parameter.
2. `MainController.set_implementations`: Adds the implementation contracts for a given `A` parameter.
3. `MainController.set_global_market_debt_ceiling`: Sets the global debt ceiling.
4. `MainController.set_market_hooks`: Sets the market hooks for a market. In case, `0x0` is passed as the market, global hooks for all markets are set. Note that one or multiple hooks can optionally be activated.
5. `MainController.set_amm_hook`: Set the optional AMM hook for a market.
6. `MainController.add_new_monetary_policy`: Adds a new monetary policy and associates it with an ID.
7. `MainController.change_existing_monetary_policy`: Replaces an ID's monetary policy. Thus, multiple markets will have their policy changed at once.
8. `MainController.change_market_monetary_policy`: Replaces the monetary policy of a market by changing the policy ID it points to.

9. `MainController.set_peg_keeper_regulator`: Sets the `PegKeeperRegulator` and may optionally migrate the `PegKeeper` contracts from the old regulator to the new one (see Peg Keeper).
10. `MarketOperator.set_amm_fee`: Sets the fee of the AMM.
11. `MarketOperator.set_amm_admin_fee`: Sets share of the AMM fee that `collect_fees` will collect for the governance.
12. `MarketOperator.set_borrowing_discounts`: Sets the loan and liquidation discounts.
13. `MarketOperator.set_liquidity_mining_hook`: Sets the liquidity mining hook (see Hooks).
14. `MarketOperator.set_debt_ceiling`: Sets the local debt ceiling for the market.
15. `MarketOperator.set_oracle`: Sets the oracle used by the AMM.

2.2.1.2 Hooks

Loan-manipulating operations will invoke market hooks that adjust the debt taken (`create_loan` and `adjust_loan`) or the amount of stablecoins burned (`close_loan` and `liquidate`) with a positive or negative delta to allow for charging extra fees or for giving out rebates. In the case of rebates, the adjustment made is limited to an allowance that can be increased by burning stablecoins with `increase_total_hook_debt_adjustment` in the `MainController`.

Further, note that when collateral is deposited or withdrawn (including soft-liquidations with `exchange`), an AMM hook, if set, is invoked. Essentially, the AMM hook will hold the ERC-20 allowance of the collateral token to pull funds out of the AMM to allow generating yield with collateral. Ultimately, before withdrawals and after deposits the hook will be invoked. The AMM hook can be set by the governance with `set_amm_hook`.

Last, similar to Curve, the AMM supports an optional liquidity mining hook during adding and removing liquidity as well as exchanging.

Note that no hooks have been part of this review as mentioned in Excluded from scope.

2.2.1.3 Monetary Policy

Monetary policies compute interest rates for markets given the current state. The `AggMonetaryPolicy2` computes the interest rate as a base rate multiplied with an exponential of e where the power is the difference between the target price and the stablecoin's price divided by some `sigma`, minus the ratio between the debt fraction of debt held by keepers and a given target fraction. Additionally, that rate is a factor depending on the utilization of a market's debt ceiling.

2.2.2 Peg Keeper

`PegKeeper` contracts can allocate funds to AMMs to stabilize the peg by providing liquidity on markets where the stablecoin is paired with other stablecoins. They are centrally managed by a `PegKeeperRegulator` where governance can call

1. `add_peg_keeper` to add a new peg keeper and set its debt ceiling,
2. `adjust_peg_keeper_debt_ceiling` to change the debt ceiling of a peg keeper,
3. `remove_peg_keeper` to remove a peg keeper (only possible when no debt is outstanding),
4. `set_worst_price_threshold` to set a global threshold for providing liquidity at too high prices,
5. `set_price_deviation` to set the global maximum deviation from the EMA oracle price of the peg keeper's pool,
6. `set_debt_parameters` to set the global parameters `alpha` and `beta` which are part of the definition of the allocation ratio,
7. and `set_killed` to pause deposits and/or withdrawals (or unpause).

Note that increasing the debt ceiling of a pegkeeper will automatically mint tokens to it while lowering the debt ceiling will try to recall the debt (see below).

With `update`, keeper bots may update a peg keeper's allocation. The process can be summarized as

1. In case not enough time has passed since the last action, do not change the liquidity provision.
2. If the peg keeper's pool has more of the paired with stablecoin than the DFM's stablecoin, provide liquidity. If it has less, withdraw liquidity. Note that the deposit and withdrawal will be one-sided (only DFM's stablecoin) and thus automatically swap to a more balanced state.
3. Assuming the stablecoins are equal in value, the LP token will have appreciated in value due to the more balanced state of the pool according to the Curve Stableswap invariant. Thus, a profit is realized. A share of the profit in LP tokens is shared with the keeper bot.

Recall that lowering the debt ceiling will try to recall the debt. That will burn the difference in the peg keeper. If an insufficient amount of stablecoins is available to the peg keeper, the full balance will be burned and the outstanding debt to be repaid will be tracked as an owed debt. Note that providing and withdrawing liquidity will try to repay as much owed as possible (balance before providing and balance after withdrawing).

Note that the provisions and withdrawals are limited (upper bound). Both are limited by 20% of the balance difference of the two assets in the pool. Additionally, provisions are also limited by the stablecoin balance of the peg keeper while withdrawals are limited by the amount invested in the pool (active debt). The regulator further provides additional limits with the following functions that are queried by the keepers:

1. `get_max_provide()`: In case deposits are killed or the aggregator reports a USD price below the peg, or the pool's oracle and spot price deviate too much, deposits are disallowed. Additionally, if the highest price of the other peg keeper pools' oracle prices is more than a given threshold smaller than the peg keeper pool's oracle price, deposits are disallowed. Otherwise, deposits are allowed. However, they are limited by a function depending on the debt utilizations of all peg keepers.
2. `get_max_withdraw()`: In case withdrawals are killed or the aggregator reports a USD price over the peg, or the pool's oracle and spot price deviate too much, withdrawals are disallowed. Otherwise, no additional limit is put on the keeper.

As described in Core, the regulator can be migrated. A migration sets the new regulator of the present peg keepers to the new one and provides the new regulator with the peg keeper addresses and ceilings. Last, governance can set the keeper bot's profit share with `set_new_caller_share()`.

Note that the aforementioned aggregator is expected to be an elaborate oracle that provides a safe price of the stablecoin in terms of the pegged asset.

2.2.3 Stablecoin

The implemented stablecoin that is intended to be used is a flashmintable ERC-20 token. Further, the token inherits from LayerZero's OFT implementation to implement cross-chain capabilities with LayerZero as the infrastructure.

2.2.4 Changes in version 2

The most notable changes in version 2 are:

1. AMM hooks were removed.
2. The market hooks have been redesigned. To summarize, hook types defining what a hook can do were introduced. The hook debt is now removed when the hook is removed.

2.2.5 Changes in version 3

The most notable changes in version 3 are:

1. The protocol can be (un)paused (`set_protocol_enabled`). Pauses loan creation and adjustments as well as fee collection.
2. The delegations can be de- or reactivated (`setDelegationEnabled`). If delegations are deactivated, a delegate cannot execute an action on behalf of a user.
3. Governance can set the above values to both true or false. A new role was introduced, `CORE_OWNER.guardian()`, that can execute the above actions to deactivate/pause.
4. `MarketViews` has been introduced as a peripheral contract. A set of view functions have been migrated there. For that, some adaptations have been made in the `MainController`. Note that the `MarketViews` assumes that bands are in a range `[x, x+5000]`.
5. The following changes have been made to the stable coin:
    1. The `transferOwnership` and `renounceOwnership` functions have been deactivated as this is handled through the `CORE_OWNER`.
    2. The OFT `_debit` and `_credit` (part of the bridging process) can be (de)activated with `setBridgeEnabled`.
    3. The flashmintable amount can be set to zero with `setFlashMintEnabled` (or reactivated again).
    4. Default options are introduced. When setting a peer, these default options are the `OAppOptionsType3.enforcedOptions` initially set when setting a peer. Default options can be set with `setDefaultOptions`. Note that the enforced options can be still modified with the `OAppOptionsType3` functionality.
    5. Further, `sendSimple` as a simplified version of `OFTCore.send` has been introduced. Similarly, `quoteSimple` is a simplified version of `OFTCore.quoteSend`. Note that the simplified sending does not include slippage protection and adding extra options, and additionally it does not support composing.
    6. Last, a set of getters has been introduced.

2.2.6 Trust Model and Roles

The system defines the following key roles:

1. User: Untrusted.
2. Governance: Fully trusted. Governance is fully trusted and may mint arbitrary stablecoins, block users from performing actions and perform other malicious actions.
3. Keepers: Untrusted but expected to use keeper functionality. Keepers will call the Peg Keeper updates, we expect that the incentives will be set such that they will be incentivized to update the peg keepers.
4. Guardian. Trusted. The guardian may DOS users from interating with the protocol by pausing the protocol or by disabling delegations. Note that the guardian may deactivate bridging and flashloans.
5. Bridge relay: Trusted. The bridge relay can set new peers on the OFT.

The governance is defined as the `CORE_OWNER` contract's owner which is out-of-scope of this review. The trust model of that contract is inherited due to the possibility of owner changes that may be the consequence of other core owner roles' actions.

The contracts require a certain setup, governance is expected to set up the system accurately and correctly. Namely, setting the oracle, monetary policies, implementations, debt ceilings and other parameters requires careful execution and decision-making. We expect that the contracts in scope will be used.

Hooks, external systems (e.g. Layer Zero) and similar are out of scope and fully trusted. Misbehaviour may lead to theft.

Tokens should be standard ERC-20 tokens with no fees, no rebasing and no other uncommon behaviour.

Further, we expect that it won't be necessary to loop over more than `MAX_SKIP_TICKS` and that all token balances are less than `2**127`. Please see other assumptions made in the context of the referenced Curve audit.

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	0

6 Resolved Findings

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

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

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

- Hook Adjustments Can Be Unbacked Code Corrected

Low-Severity Findings	14

- Bad Interface Used in MarketViews Code Corrected
- Hook Debt Increase Code Corrected
- AMM Migration Code Corrected
- Decreasing Hook Adjustments Code Corrected
- Errors in Controller Simulations Code Corrected
- Failure of Returning Market Data Code Corrected
- Flashmint Breaks Previous Assumptions Code Corrected
- Inconsistent Parameter Validation Governance Functions Code Corrected
- Interface Incompleteness Code Corrected
- NG Pools Are Not Supported by PegKeepers Code Corrected
- Reentrancy Locks Specification Changed
- Remaining Mintable Wrong Code Corrected
- Rounding up Code Corrected
- _price_in_range Can Suffer From Over/Underflow Code Corrected

Informational Findings	15

- PegKeeperRegulator.aggregator Could Be Immutable Code Corrected
- Error Message Can Be Misleading Code Corrected
- Event Improvements Code Corrected
- Getter Lacks Dynamic Fee Code Corrected
- Inconsistent Rate Updates Code Corrected
- Log Precision and Efficiency Could Be Increased Code Corrected
- max_borrowable Might Return Incorrect Values Code Corrected
- No MAX_RATE Limitation for Monetary Policies Code Corrected
- Target Debt Fraction Zero Code Corrected
- Overflow With Threshold Specification Changed
- Sanity Check off by One Code Corrected
- Specification Mismatches Code Corrected

- Unused Variables Code Corrected
- Built-in Min Value Can Be Used Code Corrected
- _call_view_hooks Is Inconsistent Code Corrected

6.1 Hook Adjustments Can Be Unbacked

Correctness Medium Version 1 Code Corrected

CS-DFM-001

Negative hook adjustments can create extra incentives for users. However, they have to be backed by earlier burns (or by earlier positive hook adjustments). However, the positive hook adjustments are also included in the fees. Thus, there is positive hook adjustments can be double-spent.

Consider the following example:

1. Assume that fees are freshly collected over all markets so that no interest is pending and so that `minted = total_debt + redeemed` (for simplicity). Assume that `total_debt = 100` and `redeemed = 100` so that `minted = 200`.
2. Alice creates a loan of `100`. The debt adjustment is set to `100`. Thus, Alice creates a loan of `200` and mints `100`. Hence, `total_debt = 300` and `minted = 300`. Note, that `hook_debt_adjustment` is increase by `100`, too.
3. Ultimately, fees are collected (with an empty market list) again and `100 = total_debt + redeemed - minted` tokens are minted.
4. Now Bob performs an action that has a negative debt adjustment. Ultimately, the rebate will be based on already spent amounts.

To summarize, it can be possible that hook debt adjustments and fees lead to double-minting so the hook limitations are not working as intended.

Code corrected:

The code has been adjusted. Now it is ensured that this double counting does not occur.

6.2 Bad Interface Used in `MarketViews`

Correctness Low Version 3 Code Corrected

CS-DFM-031

The newly introduced `MarketViews` tries to call `AMM.collateral_balance` as defined in the interface. However, that function has been removed from the AMM as of Version 2.

Code corrected:

The code now uses the `ERC20.balanceOf` function.

6.3 Hook Debt Increase

Correctness Low Version 2 Code Corrected

CS-DFM-032

The `increase_hook_debt` function changes the fees currently available to the system which may lead to a loss of debt adjustments.

Consider the following example:

1. Assume that `minted = 1000`, `redeemed = 500`, `total_debt = 1500` and `total_hook_debt = 0`.
2. Note that this implies that `1000 = 1500 + 500 - 0 - 1000` fees are available.
3. Alice increases the hook debt. Now, `total_hook_debt = 200`.
4. That implies that `800 = 1500 + 500 - 200 - 1000` fees are available. 200 has become unavailable as fees.
5. Now, consider that the hook is removed. That would add 200 back. However, 200 would ultimately be lost.

Due to violating the property that an increase of the debt hook should not affect the fees, tokens are burned and not collected as fees. It is possible for attackers to withold fees from the fee recipient.

Note that Alice could also increase the hook debt by more, using tokens being external of this accounting, leading to an underflow that would DOS `collect_fees` for a while. (Imagine here having a `total_hook_debt = 1000`)

Code corrected:

The code has been corrected by increasing `redeemed` by the increase amount.

6.4 AMM Migration

Design | Low | Version 1 | Code Corrected

CS-DFM-002

Setting AMM hooks might lead to dangerous scenarios. Especially, since there is a strict equality in the comparison of pre- and post-collateral balances.

```
assert AMM(amm).collateral_balance() == amount, "DFM:C balance changed"
```

Consider, the following scenarios:

1. The hook invests WETH in stETH. Due to the rebasing nature of stETH, transfers are not accurate. Thus, the new collateral balance of the AMM will slightly decrease. However, that could be fixed with simple donations.
2. Governance wants to update the hook. But someone donated pre-execution a small amount to the hook. That may DOS governance actions. However, that can be fixed in different ways.
3. There is an exit fee in an external system used. That cannot be handled.
4. A hook's collateral balance function could be broken and return wrong results. Technically, that could be also fixed by setting an intermediate exchange hook with a temporarily broken mechanism and fixing later on (e.g. upgrade, storage variable or similar).

Ultimately, the strict equality may lead to migration problems. The mechanism can be dangerous - especially due to the strict equality.

Code corrected:

Amm hooks were removed.

6.5 Decreasing Hook Adjustments

Design Low Version 1 Code Corrected

CS-DFM-003

The hook debt adjustments can be increased by arbitrary parties. However, they can be decreased only by hooks. In case the governance decides to deactivate hooks, tokens could be unretrievable. Thus, there is a lack of decreasing options for hook adjustments.

Code corrected:

The hook adjustment can be reduced by removing the hook.

6.6 Errors in Controller Simulations

Correctness Low Version 1 Code Corrected

CS-DFM-004

The MainController's `get_close_loan_amounts()` and `get_liquidation_amounts()` incorrectly return an adjusted amount of debt repaid. Namely, both functions compute `total_debt_repaid` as `debt + hook_adjustment`. However, in both the closing of loans and the adjustment of loans, the hook adjustment is used to increase/decrease the burned amount. The repaid debt is not affected by the hook adjustment.

Code corrected:

The code now correctly returns the data.

6.7 Failure of Returning Market Data

Correctness Low Version 1 Code Corrected

CS-DFM-005

The `get_market()` and `get_amm()` functions in the `MainController` try to return `0x0` for indexes that are out of the array's bounds.

```
if i > len(self.collateral_markets[collateral]):
    return empty(address)
```

However, if `i == len(...)`, the array is also accessed out of bounds.

Code corrected:

Code corrected by changing the condition to `i >= len(...)`.

6.8 Flashmint Breaks Previous Assumptions

Design | Low | Version 1 | Code Corrected

CS-DFM-006

In our previous audit of Curve stablecoin, the assumption is made that the total balances are smaller than `2**127`. This assumption is broken by the Flashmint function, which can mint an arbitrary amount of tokens.

Code corrected:

`maxFlashLoan` was overridden to be limited to `2**127 - totalSupply()`.

6.9 Inconsistent Parameter Validation Governance Functions

Design | Low | Version 1 | Code Corrected

CS-DFM-007

The `market` parameter passed to `set_market_hooks()` and `change_market_monetary_policy()` are not validated to be existing markets. That is inconsistent with other governance functions which do not allow operating on non-existing markets.

Code corrected:

A new internal function `_assert_market_exists()` was added to validate the market parameter. This function is called from `set_market_hooks()` (more specifically, the functions that replaced `set_market_hooks()`) and `change_market_monetary_policy()` to ensure the market exists before proceeding.

6.10 Interface Incompleteness

Correctness | Low | Version 1 | Code Corrected

CS-DFM-008

The interface `ICoreOwner` does not have the method `feeReceiver` that is used in the implementation of the `MarketController` and the `PegKeeper`.

Code corrected:

`ICoreOwner` was deleted and replaced by `IProtocolCore`. This new interface implements required methods for all contract in scope.

6.11 NG Pools Are Not Supported by PegKeepers

Correctness | Low | Version 1 | Code Corrected

CS-DFM-009

The supported signatures supported by PegKeepers are using `uint256[2]` whereas `DynArray[uint256, 8]` is used for the CurveStableSwapNG pools.

The PegKeepers are in consequence only supporting old pools types.

Code corrected:

The Curve interface was updated to support the CurveStableSwapNG pools.

6.12 Reentrancy Locks

Design | Low | Version 1 | Specification Changed

CS-DFM-010

Some external calls to the collateral token deviate from the checks-effects-interactions pattern. If no reentrancy lock is present, these calls might introduce reentrancy possibilities (especially read reentrancies) before the state is fully updated. We could not find a case where the non-updated state might be relevant information. Still, it might be worth considering fully adhering to the checks-effects-interactions pattern.

Further, the reentrancy locks appear to be set inconsistently. We cannot see the underlying logic of how they are added. For example, some governance setters have a nonreentrant decorator and some do not.

Please see 6.1 of our crvUSD report.

Specification changed:

DeFi.Money specified that no token and no hook will have callbacks to untrusted addresses.

6.13 Remaining Mintable Wrong

Correctness | Low | Version 1 | Code Corrected

CS-DFM-011

`MainController.get_market_states()` computes and aggregates data of input markets. However, when computing the amount that is mintable based on the global debt ceiling and global debt, the pending interest is not included. Thus, the estimation of the remaining mintable amount could be off. Note that the interest is accounted for in the market-specific mintable amount.

Code corrected:

The getter now includes the pending interest for a given market when checking against the global debt ceiling.

6.14 Rounding up

Correctness | Low | Version 1 | Code Corrected

CS-DFM-012

The `MarketOperator.min_collateral` should round up to the nearest integer to ensure that the returned value is not below the minimum. For reference, note that Curve implements rounding up.

Further, note that rounding of debt is not done in favor of the system. In contrast, Curve rounds the debt of users in favour of the system. As a consequence, for DFM Core, `sum(user_debt) < total_debt` can also hold in certain situations. That may not allow for graceful exits of the system and may leave some dust in `total_debt`. In contrast, Curve ensures that `sum(user_debt) >= total_debt` (rounding up errors, however requiring to handle `user_debt > total_debt`) on repays.

Code corrected:

The function `MarketOperator.min_collateral` has been updated to round up to the nearest integer.

The case where `sum(user_debt) < total_debt` holds is solved by adding this dust to the `debt_adjustement` variable if the last user is repaying (or being liquidated). This ensures that the total debt is always zeroed out when no more users are in the system.

6.15 `_price_in_range` Can Suffer From Over/Underflow

Correctness **Low** **Version 1** **Code Corrected**

CS-DFM-013

`PegKeeperRegulator._price_in_range` can suffer from over/underflow with potential price manipulation in the following code:

```
return unsafe_sub(unsafe_add(deviation, _p0), _p1) < deviation << 1
```

Assume that the `deviation` is `1000`. Set the `_p0` to `max_value(uint256)` and `_p1` to `10` or visa versa, the result will be `True` because of the over/underflow. Note that due to flashminting high amounts such prices might be possible.

Code corrected:

This is no longer possible since the flashminting feature has been reduced to fit the max balance assumption of `2**127`.

6.16 Built-in Min Value Can Be Used

Informational **Version 1** **Code Corrected**

CS-DFM-014

In `MainController._adjust_loan_bounds()`, `-2**255` is used to define the min value, for consistency with the max value, it can be replaced with `min_value(int256)`.

Code corrected:

The code has been adjusted accordingly.

6.17 Error Message Can Be Misleading

Informational | Version 1 | Code Corrected

CS-DFM-015

In the `MainController.get_pending_market_state_for_account()`, the first assertion checks the following condition: `convert(debt, int256) + debt_change > 0`. This assertion is correct however the error message can be misleading since the sum can be zero and it will still fail. The error message should be updated to reflect this.

Code corrected:

The error message has been adjusted to reflect that the value is non-positive.

6.18 Event Improvements

Informational | Version 1 | Code Corrected

CS-DFM-016

Some events can be improved:

1. The `SetImplementations` event in the `MainController` does not log for which `A` the new implementations are set. This is a problem because the `MainController` can have different `A`-s and it is not clear for which `A` the new implementations are set.
2. When a loan is manipulated (through the `MainController`), three different events can be emitted (`CreateLoan`, `AdjustLoan`, `CloseLoan`). However, none of these events include the address of the caller (not necessarily the same as the account), which obfuscates the origin of the manipulation (delegation or not).

Code corrected:

The events emit the additional data.

6.19 Getter Lacks Dynamic Fee

Informational | Version 1 | Code Corrected

CS-DFM-018

`get_amount_for_price` computes the amount that needs to be exchanged to reach a given price in the `AMM`. However, the computation lacks the dynamic fee and thus might be imprecise. In contrast, Curve includes an estimation with the dynamic fee.

Code corrected:

The code has been adjusted to include the dynamic fee and corrsponds now to the Curve implementation.

6.20 Inconsistent Rate Updates

Informational | Version 1 | Code Corrected

CS-DFM-019

The loan-adjusting functions inconsistently update the rate. Namely, the `_update_rate()` occurs typically after `minted` or `redeemed` has been updated. However, for `adjust_loan()` and `collect_fees` that occurs earlier. Thus, there is an inconsistency in execution. While that has no impact on the monetary policy in scope, future monetary policies could rely on correct values (should rely on consistent updates of these).

Note that mints, burns and transfers typically occur after the rate updates. The rate will not have the ability to update according to ERC20 balances reliably.

Code corrected:

The code has been adjusted to update the rate always as the last operation. Note that in any case it is not expected that data of hooks is relevant for the rate updates (e.g. increase and decrease of hook debt).

6.21 Log Precision and Efficiency Could Be Increased

Informational | Version 1 | Code Corrected

CS-DFM-020

Curve upgraded the `log2` to `logn` for log calculations. This makes the log calculations more consistent between the `AMM` (already using `logn`) and the `MarketOperator` (using `log2`) but also makes it more gas efficient.

Code corrected:

The code is corrected and the log calculations were upgraded to `logn`.

6.22 No MAX_RATE Limitation for Monetary Policies

Informational | Version 1 | Code Corrected

CS-DFM-022

The monetary policies could return unexpectedly high values. Limiting these values in the main controller could help reduce the impact of errors in future monetary policies.

Note, that the forked code of code has a `MAX_RATE` parameter that limits the value returned by the monetary policies.

Code corrected:

The rate updates done by the `MainController` now validate against a `MAX_RATE`.

6.23 Overflow With Threshold

Informational | Version 1 | Specification Changed

CS-DFM-023

The peg keeper regulator disables liquidity provisions as follows:

```
if largest_price < unsafe_sub(price, self.worst_price_threshold):
    return 0
```

Note that `price` could be very low in case of a severe depeg. While not providing liquidity in such cases might be reasonable, the behaviour of `worst_price_threshold` is undocumented.

Specification Changed:

An explanatory comment has been provided. In case of an underflow, no liquidity will be provided.

6.24 Sanity Check off by One

Informational | Version 1 | Code Corrected

CS-DFM-024

`add_market()` enforces that

```
assert admin_fee < MAX_ADMIN_FEE
```

Note that 100% however would be a legitimate value. That would make the initialization of a market consistent with the market operator's `set_amm_admin_fee()` function.

Code corrected:

The check validates now that it is <= instead of <.

6.25 Specification Mismatches

Informational | Version 1 | Code Corrected

CS-DFM-025

The functionality is documented as NatSpec or as comments. The following errors and imprecise descriptions were uncovered:

1. `get_market` and `get_amm` specify:

   ```
   Iterate over several amms for collateral if needed
   ```

   However, they do not iterate but only access the i-th element.

2. The NatSpec of `get_market_states_for_account` specifies:

   ```
   Account health (liquidation is possible at 0),
   ```

   However, liquidations are possible at `health < 0`.

3. A comment in `collect_fees` specifies:

   ```
   # Difference between to_be_redeemed and minted amount is exactly due to interest charged
   ```

   However, technically, the fee could be also charged due to a hook.
4. A comment in `AMM.__init__` specifies:

   ```
   # sqrt(A / (A - 1)) - needs to be pre-calculated externally
   ```

   However, this is not the case anymore.
5. A comment in `AMM.__init__` specifies:

   ```
   # 18 decimals: math.log2(10**10) == 59.7
   ```

   However, the log is of `10**18`.
6. The specification of `AggMonetaryPolicy2.TARGET_REMAINDER` specifies that the rate is scaled by two at 90% utilization of the debt ceiling. However, it is scaled by a factor of 1.9.

Code corrected: All the above issues were corrected.

6.26 Target Debt Fraction Zero

Informational | Version 1 | Code Corrected

CS-DFM-026

The constructor of the monetary policy allows `target_debt_fraction = 0`. However, that may lead to divisions-by-zero in `calculate_rate`. Furthermore, it could make it would make it more consistent with the setter of the target debt fraction.

Code corrected:

The code has been adapted to ensure that `target_debt_fraction > 0`.

6.27 Unused Variables

Informational | Version 1 | Code Corrected

CS-DFM-027

The following variables are defined but not used in the code:

1. `MarketOperator.STABLECOIN`
2. `MarketOperator.MAX_RATE`
3. `PegKeeper.ADMIN_ACTIONS_DELAY`
4. `MainController.MAX_ACTIVE_BAND`

Also, note that `AMM.BORROWED_PRECISION` is used and defined but has no effect and, thus, increases the size of the bytecode unnecessarily.

Code corrected:

The unused variables have been removed. The borrowed precision in the AMM is not used anymore.

6.28 `PegKeeperRegulator.aggregator` Could Be Immutable

Informational | Version 1 | Code Corrected

CS-DFM-028

`PegKeeperRegulator.aggregator` could be immutable since it's never updated after initialization.

Code corrected:

The aggregator is now an `immutable`.

6.29 `_call_view_hooks` Is Inconsistent

Informational | Version 1 | Code Corrected

CS-DFM-029

`_call_view_hooks` is inconsistent with `_call_hooks`. Namely, the latter asserts the bounds and limits only when the adjustment is non-zero. Even though the result will be the same, having the same execution flow for both functions may help improve maintainability and consistency.

Code corrected:

The code has been adjusted to be more consistent.

6.30 `max_borrowable` Might Return Incorrect Values

Informational | Version 1 | Code Corrected

CS-DFM-030

Since `MarketOperator.max_borrowable` doesn't check if `n_bands` is in the range of `MIN/MAX _TICKS`, it will return non-zero values for `n_bands` that are out of the range, even though the market does not support them.

Code corrected:

The code has been corrected to only accept valid `n_bands` values.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Events Not Emitted

Informational | Version 1 | Code Partially Corrected | Acknowledged

CS-DFM-017

Event emissions can help retrieve data. While the smart contracts typically emit events, the events are not always emitted. Below is a (potentially incomplete) list where logging is missing:

1. `MainController.__init__`: Adds monetary policies but does not emit `AddMonetaryPolicy`.
2. `MarketOperator.initialize`: Does not emit any relevant event.
3. `AMM.initialize`: Does not emit any relevant event.
4. `StableCoin.setMinter`: Does not emit any relevant event.

Code partially corrected:

All but 3. have been implemented. Note that the `MainController` when initializing also does not the events for the AMM initialization values.

Acknowledged::

DeFi.Money acknowledged 3.

7.2 NatSpec Problems

Informational | Version 1 | Code Partially Corrected | Acknowledged

CS-DFM-021

NatSpec can help document the code. While the codebase has quite a lot of NatSpec, much is missing. Note that NatSpec can also help frontends such as Etherscan show details about the functions.

Below is a list of examples of missing NatSpec or NatSpec errors:

1. `setDelegateApproval` lacks parameter documentation.
2. Similarly, that is the case for `change_existing_monetary_policy`.
3. `stored_admin_fees` does not define an `@return`.
4. `_calculate_debt_n1` does not include `amm` and `price`.
5. `adjust_loan` in `MarketOperator` is missing.

Note that these examples are not a complete list. We would highly encourage to check for more occurrences (and complete the NatSpec at least for the external-facing functions).

Code partially corrected:

The NatSpec documentation has been improved. Nevertheless, some NatSpec is incomplete.

Acknowledged:

DeFi.Money acknowledged the above.

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 Batched Getters Get Results for Independent Executions

Note Version 1

Users of public getters should be aware that the getters such as `get_market_states()` do not consider the actions to be sequentially executed. More formally, the iterations are independent of each other and do not affect each other.

Similarly, that is the case for `get_liquidation_amounts()` where the hook debt adjustments could be depending on the previous iterations. Thus, a user should not expect the result to be the same when the same batch is liquidated iteratively.

8.2 Bridge Pausing

Note Version 1

Note that the OFT bridging functionality can be paused. Pausing should be always be considered carefully due to the possibility of pending messages.

Further, note that the `view` functions are not pause (e.g. quoting functions) and will return values under the assumption that the bridge is not paused.

8.3 Changing the Address of Monetary Policy

Note Version 1

Note that due to gas reasons, the rate is not updated on AMMs when the address that a `mp_idx` points is changed. The consequence is, that the old rate will be used. The new monetary policy will be actually used for a market after the first interaction with a market or when collecting fees (note that NatSpec recommends doing this, too).

8.4 Changing the Implementation for an A

Note Version 1

Users should be aware that the implementation can change for an A. However, that does not impact the implementation of already deployed markets.

8.5 Debt Ceilings

Note Version 1

A debt ceiling is put in place for each market. Additionally, a global debt ceiling is present that works across all markets. It's worth noting that these ceilings are not strict limits. Namely, both ceiling types may be exceeded due to interest generated. For the global ceiling, the interest generated by all markets will have an impact on the ceiled debt. Note that the codebase does not account for the interest of all markets when trying to bound the debt ceiling (that is expected due to gas).

8.6 Insufficient Validation

Note Version 1

Note when setting new implementations the A is validated. Similarly, when setting the oracle the read price and the written price are ensured to be equal. However, users and governance should be aware that these checks are helpful sanity checks but do not guarantee the correctness of these contracts in the future.

8.7 Jumps in Monetary Policy

Note Version 1

Users and governance should be aware that the monetary policy might have high jumps in interest in case of infrequent updates. More specifically:

1. The monetary policy depends on the total market debt. However, that value might be outdated in case some market has not been touched for a long time.
2. The monetary policy depends on the stablecoin price. If the price changes between rate updates, the interest rate might have high jumps.

In case all markets are active, the interest rate should be relatively smooth for the case mentioned above.

Also jumps in rate can happen every time a new monetary policy is set. This is hardly predictable for a user, but he must be aware that the rate could change at any time.

8.8 Market Hook Behaviour and Impact

Note Version 1

The market and global hooks' return values are limited to safe amounts for the protocol. However, the hooks can have a great impact on the execution flows of the smart contracts. Below is a list of considerations:

1. Hooks can increase the debt users and charge very much.
2. Hooks can make the protocol unsafe by reverting (e.g. blocking liquidations).
3. Hooks can make weird execution paths more profitable (e.g. repay 50% of debt and then close the loan could be the better option than just closing the loan).
4. The view-hooks should always return the value that the write-hooks return.

Governance should carefully design the hooks. Note that the list above might be incomplete.

8.9 Oracle Considerations

Note Version 1

Oracles should be secure. Insecure oracles or bad ones may lead to breaking assumptions (as in Curve).

Note that besides the Curve assumptions, one has been added. Namely, the non-view price function should always return the same value as the view function (in any order in the same trace of an entry-point of the system).

8.10 Pausing Main Controller

Note Version 1

`close_loan` and `liquidate` cannot be paused as these reduce the debt. However, note that `adjust_loan` could be technically also allowed when only the debt is reduced.

8.11 Peg Keeper Considerations

Note Version 1

The pegkeeping mechanism tries to allocate liquidity to pools with other stablecoins so that liquidity is provided at the peg. Users and governance should be aware of the following behaviour:

- The market conditions may not allow to withdraw the provided liquidity (or to ever provide liquidity). Note that this implies that the peg keepers **may not be removable** (due to the non-zero debt checks).
- The regulator **does not try to steer peg keepers to withdrawals**, if the debt ceiling has been reduced and owed debt is outstanding.
- With the migration of pegkeepers to a new regulator, the behaviour of the mechanism may significantly change.
- Governance should **assign the stablecoin minter role** to the new regulator **and remove it** from the old one when a regulator migration occurs.

Further, note that the other stablecoins might lose their peg which may impact the peg.