Reflexer

GEB Protocol Audit

OpenZeppelin | security

The GEB protocol built by the Reflexer Labs team, is a stablecoin project based on the core design principles of the Maker's Multi-Collateral Dai (MCD) system. It brings new, exclusive features and is aimed to improve efficiency and user experience.

Reflexer is made up of several modules hosted in several repositories. This audit covers the core contracts of the protocol present in the `src` directory of the `geb` repository. In particular, we audited commit `261407b6b332c2063e4256aa5f9b223d52dad7e1` and excluded the internal `test` directory from the scope. Relevant modules that interact with this core system but that are left out of scope include the governance mechanism and the oracle price feed system.

Update: *Reflexer Labs has fixed in individual pull requests the issues we reported. We refer to these updates in their corresponding issues. Our analysis of the mitigations assumes the pull requests will be merged, but disregards any other potential changes to the code base.*

System overview

The main goal of the protocol is to provide an asset to be used in other DeFi protocols, which is called reflex index, which is less volatile than normal crypto assets.

To obtain the system asset, users can open `SAFE` positions (similar to collateral debt positions in MCD). These require a deposit of an amount of collateral, whether it is an ERC20 token or native ether, defined by an established safe collateralization ratio.

Once opened, `SAFE` positions give the opportunity for users to take out their stable reflex index from the system in the form of an ERC20 token and use it in other protocols. Whenever an user deposits collateral and opens a `SAFE` position, debt backed by that collateral is generated alongside the creation of the reflex index.

Users will also pay stability fees associated with their `SAFE` positions. Stability fees are then sent to tax receivers like the `AccountingEngine` contract, which is in charge of starting debt and surplus auctions, or to secondary tax receivers like the

`StabilityFeeTreasury` contract, which is a fund reserve used to finance operations and eventual system costs.

To achieve stability, the reflex index is managed by two different prices, a market price given by an oracle price feed, and a redemption price accounted internally in the system. The internal redemption price relies on a target initial value (1 USD peg in this case) and on a redemption rate that modifies the internal price value in a way that goes against market price fluctuations.

In particular:

- Whenever the market price is higher than the internal redemption price, the incentive is to create more `SAFE` positions to mint more reflex indexes and sell them to balance out supply/demand in the market, thereby lowering the market price. In this sense, lowering the redemption rate will make debt cheaper and users can benefit from doing arbitrage between the two prices.

- Whenever the market price is lower than the internal redemption price, the incentive is to reduce debt in `SAFE` positions, which now is more expensive, exposing undercollateralized users to liquidation. Users can escape liquidations, by buying cheaper reflex indexes on the market and adjust their collateralization ratio to a safe level, all the while increasing market price.

In the case a `SAFE` becomes under collateralized, the position can be liquidated and their collateral confiscated to be auctioned. Reflexer introduces the possibility for users to add insurance for their positions, by assigning a governance-whitelisted saviour contract that will be automatically called and asked for more collateral whenever a specific position is being liquidated. In this way, users can protect themselves by depositing backup collateral in their chosen saviour contract. Auctioned collateral aims to obtain back as much reflex indexes as possible to cover debt.

In the scope we audited both English and Fixed-Discount kinds of collateral auctions. English auctions are made of two phases where users first compete with an increasing bid amount of reflex indexes and later with a decreasing amount of collateral to buy back from the auction. When there is an intense network usage, users can struggle in finding some blockspace to include their bids in the chain, and expert users can benefit from it being the only one placing the smallest bid and winning auctions during the network congestion.

With Fixed-Discount options there is only one phase where users buy collateral at a fixed discount price. In this way, the problem of competing against bidders during network congestion is removed, and also flash loans can be used to buy all the collateral and close the auction in one single transaction, without actually needing to have the reflex indexes upfront.

Fixed-Discount auctions are one of the differences with respect to MCD. A complete list of differences between the two systems can be found in the documentation.

If there is some debt left or debt which is not backed by any asset, debt auctions can be started. Debt auctions are meant to offer protocol tokens, another asset of GEB protocol which is similar to the MKR token. The goal is to dilute the protocol token supply by minting new tokens and selling them in exchange for reflex indexes.

In the same way, if there is any surplus of reflex indexes in the system, this can be auctioned asking for protocol tokens which are then burned. In this sense, debt and surplus auctions aim to control protocol token supply by getting rid of debt and surplus inside the system.

Lastly, like MCD, Reflexer offers a contract to gain interests over deposits. This is the `CoinSavingsAccount`, where users can deposit their reflex indexes and earn interests over time. Matured interest is accounted as unbacked debt in the system and this must be correctly covered either by auctions or surplus that settles it.

Assumptions

Since in-scope contracts are just part of the entire system, some important governance pieces are left out. For this, during the audit, the following assumptions were considered:

- Valid authorized accounts will be set during the deployment phase to give correct permissions to all the needed contracts and no more addresses will be set afterward. Moreover, addresses that don't need special rights after deployment should renounce their authorization once the system is set up.

- Oracles are working all the time with no downtime. If this is not the case there could be dangerous effects in the system like auctions that revert whenever the oracle returns a null price or if an incorrect price is returned when doing a global settlement. In this last case, the `finalCoinPerCollateralPrice` would be set to that incorrect value and this can affect following settlement operations.

- Governance will correctly set values in all the `modifyParameters` functions. These functions are key for the system because they permit to tune system parameters. Any incorrect value can bring dangerous and unexpected results or vulnerabilities.

- Governance will conduct deep analysis and audit saviour contracts code whenever they want to whitelist a new one. Malicious saviours contracts can affect the system parameters and normal system operations.

- Fixed-Discount auctions will be chosen instead of English auctions. This is to avoid network congestion dependency and to benefit from flash loans, improving user experience and system efficiency.

- Fee collection is a manual process that will be performed regularly by users or bots.

- `_struct` or a new version of the `sort` function within `LinkedList` will be implemented correctly.

- When withdrawing from the `CoinSavingsAccount` contract , users are well informed about the fact that the `updateAccumulatedRate` is not called internally and must be manually called beforehand otherwise the withdrawal amount can be less than expected.

Summary

We appreciate the efforts put into translating and making the MCD codebase more understandable, since this can help the community to gain confidence with such systems and it improves transparency and auditability of the project itself.

However, there is a generalized use of repeated code that increases complexity of the source code and makes it more difficult to understand.

Some key pieces are lacking any comments and several functions can be replaced by already tested and audited libraries. We think that the effort of translating MCD code to a more understandable wording can be even more valuable if the code base is also refactored, polished and commented where necessary. This will not only improve readability, but also gas consumption and attack surface reduction.

We reviewed the code with three auditors over the course of four weeks. Here we present our findings.

Critical severity

None.

High severity

[H01] `fastTrackAuction` accounts for debt incorrectly

Currently, when the system is shutdown, if there are any collateral auctions still occurring they can be ended by calling `fastTrackAuction`. This function is designed to end the auction and return unsold collateral and unsettled debt from the auction to the original SAFE owner. Notice that in english type auctions, the unsold collateral is the initial amount, since english auctions can be terminated prematurely only in the first phase, where no collateral is sold. On the contrary, fixed-discount type of auctions sell part of the initial collateral whenever a user calls the `buyCollateral()` function. For this reason, unsold collateral in this case can be lower than the initial amount.

A SAFE owner will receive back only the amount of collateral that was not sold, `collateralToSell`. However, the debt they receive back, which is originally taken from the SAFE when liquidation starts, will end up being the same amount as was initially taken. This means that when a SAFE's collateral auction is fast-tracked, they lose collateral but retain the same amount of debt.

This has the wider effect that SAFEs become even more under-collateralized than they were before. When `processSAFE` is called, these SAFEs will artificially increase `collateralShortfall` more than they should. In the same way, the `collateralTotalDebt` is incorrectly increased. This will have effects also in the calculation of `redemptionAdjustedDebt` which is accounted together with `collateralShortfall` to calculate the final `collateralCashPrice`. This will result in an skewed value of `collateralCashPrice`, potentially lowering the value of all coins upon redemption.

Consider changing line 298 to use the difference between the values of `bids[id].amountToRaise` and `bids[id].raisedAmount` in place of `amountToRaise`. These values may need to be fetched with the other auction parameters, since auction data will be deleted within the call to `terminateAuctionPrematurely`. This will correct the issue of `debt_` being too high on line 301, which currently causes the liquidated SAFE to receive the same `.generatedDebt` as when it started, even when an auction has sold some collateral. Note that this problem also results in erroneous accounting for the `debtBalance` of `AccountingEngine` and `globalUnbackedDebt`.

***Update**: Fixed in pull request #74.*

Medium severity

[M01] Starting debt auctions can be prevented

Within the `AccountingEngine` contract, the function `auctionDebt` contains a check that the `AccountingEngine` contract instance has no system coin balance. However, the `transferInternalCoins` function within the `SAFEEngine` contract allows transfers of internal tokens to any contract, provided `canModifySafe(src, msg.sender)` returns true. Thus, any user can transfer their internal coins to the `AccountingEngine` contract. For any nonzero amount of coins, this will cause the check within `auctionDebt` to fail.

The result of this is that any user can front-run calls to `auctionDebt` to prevent them from executing, effectively preventing any new debt auctions from starting. However, it should be noted that eventually, the front-runner may fail in their front-running attempt, and at that point a new debt auction can begin.

Consider replacing this check with a call to `settleDebt(safeEngine.coinBalance(address(this))` before the balance check is enforced. This may require changing the visibility of `settleDebt` to `public`.

Update: *Fixed in pull request #75.*

[M02] English auction bidder can win with low `bidAmount` on network congestion

English-like auctions differs from fixed-discount ones because the amount that has to be sold is auctioned among bidders competing to offer more system coins (in a first phase) for less and less collateral to buy (during a second phase).

In this kind of auction, there are two important parameters to take into account:

- bidExpiry: the time after which a bid can be settled (by contract set to 3 hours after the bid is placed).
- auctionDeadline: the amount of time an auction can last.

Whenever a bid expires or the auction deadline has passed, anyone can call the `settleAuction` function and terminate it.

Moreover, the very first bid can be any small `bidAmount > 0` and any consecutive bid must be 1.05x times greater than the previous one. The reason why the first bid can be whatever positive amount is because collateral auctions are started from the `LiquidationEngine` with an initial `bidAmount = 0`.

During network congestion, bidders' transactions may not be processed, allowing very low priced bids to be filled with no counterparty bidders competing against them. Since we understand that the Reflexer team intends to implement fixed-discount auctions, we list this issue to make the risks of English style auctions clear. Consider implementing fixed-discount auctions to alleviate this problem. Make sure to implement appropriate tests simulating high network congestion. Additionally, to avoid users bidding with tiny amounts consider establishing a non-zero minimum value for the first bid.

Update: *Acknowledged. Reflexer Labs' statement for the issue:*

> We know about the problem and we plan to use fixed discount auctions in RAI's case.

[M03] Too high `totalAuctionLength` prevents settling of fixed-discount collateral auctions

Within `FixedDiscountCollateralAuctionHouse`, the variable `totalAuctionLength` is initialized with a value of 10 years. This value is used at the start of an auction to determine the auction's deadline.

We understand that `totalAuctionLength` does not have the same meaning in fixed-discount auctions as it has in the english auction, since a fixed-discount auction can be completed using flash loans in one single transaction, thereby speeding up the duration of the auction. Having a deadline far in the future prevents a fixed-discount auction from settling.

Moreover, the `settleAuction` function is performing the same operations as lines 839-843 of the `buyCollateral` function.

If the `settleAuction` function is needed in any circumstance other than when all collateral has been sold, as in the `buyCollateral` function, consider giving a proper value to `totalAuctionLength`, so it can be called in a more reasonable timeframe. Note that the `settleAuction` function is declared as `external` and can be called by anyone when `now` is greater than the deadline.

Update: *Fixed in commit `c582fb57c746e36ed6f43ca80a7816e751c0ae2d`. The Reflexer Labs team has removed functionality from the `settleAuction` function within `FixedDiscountCollateralAuction` and made `totalAuctionLength` and `auctionDeadline` obsolete for fixed discount auctions.*

[M04] Starting surplus auctions can be prevented

Accumulated surplus of reflex indexes can be auctioned to retrieve protocol tokens to be burned. This happens in the `auctionSurplus()` function of the `AccountingEngine` contract.

The code uses a queue of debt blocks to handle debt, where new debt is always pushed into the queue and it is processed in blocks that are popped out from the queue. This is done to provide some delay in which the contract can accumulate eventual surplus which can be used to settle such debt.

In line 286 of the `auctionSurplus()` function, the code is checking whether the contract has some unactioned and unqueued debt. If the contract actually has an `unqueuedUnauctionedDebt() != 0` it should use the surplus to settle that debt completely or partially.

The problem resides in the fact that any call to the `auctionSurplus()` function can be frontrun and forced to fail by popping a debt block out of the queue and reverting the check in line 286. This can be done by an attacker by calling the `popDebtFromQueue()` function in the `AccountingEngine` contract.

An attacker can conduct this attack several times with the limit given by the size of the queue itself. In this sense the potential attack can't last forever, thereby reducing the severity of this issue. By doing this, an attacker can freeze surplus auctions and influence the system's operation.

An easy solution would be to call the `settleDebt()` function at the beginning of the `auctionSurplus()` so that any unqueued and unauctioned debt can be settled before sending any surplus to the auction contract. This would also eliminate the attack vector and safeguard system operations.

Considering calling `settleDebt()` inside `auctionSurplus()` before checking that the unqueued and unauctioned debt is zero.

Update: *Fixed in pull request #79. Upon review, the OpenZeppelin team noticed that a similar vector for frontrunning is calling the* `updateAccumulatedRate` *function of* `CoinSavingsAccount`*, which increases the debt balance of* `AccountingEngine`*. This is also mitigated by the fix from pull request #79.*

[M05] Unnecessary input parameters

In some functions, input parameters have only one acceptable value. If the input parameter does not have this value, the function call will revert.

For example, in `SuplusAuctionHouse.increaseBidSize`, the input parameter `amountToBuy` is needed. However, a strict equality to `bids[id].amountToSell` is required.

Additionally, in `StabilityFeeTreasury.pullFunds`, the `token` parameter is required to be equal to `systemCoin`.

`SurplusAuctionHouse.startAuction` has the input parameter `initialBid`, but this is always `0` when the function is called from other parts of the code.

Consider removing these unnecessary input parameters, and instead enforcing these strict equalities within the function's logic. This will improve user experience for publicly callable functions by lowering the likelihood of a transaction to revert, and simplifying the user interface. This will also assist auditors and future developers in understanding the intent of the code. If these input parameters are desired to be kept, consider explaining why in the docstrings for the function.

Update: *Acknowledged, and will not fix. Reflexer Labs' statement for this issue:*

> Maker wanted a general interface for `increaseBidSize` and `startAuction` because they're probably thinking about the future where the implementation may evolve so we'd like to keep them as they are right now. As for `StabilityFeeTreasury.pullFunds` we want to keep token because the `pullFunds(address dstAccount, address token, uint256 wad)` signature will be used in a second iteration of a treasury that can handle any type of token. This way we keep a shared interface.

[M06] Unsafe casting

In line 554 of the `TaxCollector` contract, the value of `coinBalance(receiver)` is an `uint`. This is cast to an `int` and then negated. However, since `uint` can store higher values than `int`, it is possible that casting from `uint` to `int` may create an overflow.

Consider verifying that the value of `coinBalance(receiver)` is within the acceptable range for negative `int` values before casting and negating. Consider using OpenZeppelin's `SafeCast` contract, which provides functions for safely casting between types.

Update: *Fixed in pull request #76.*

[M07] Unsafe division in `rdivide` and `wdivide` functions

The function `rdivide` on line 227 and the function `wdivide` on line 230 of the `GlobalSettlement` contract, accept the divisor `y` as an input parameter. However, these functions do not check if the value of `y` is `0`. If that is the case, the call will revert due to the division by zero error.

Other occurrences of unsafe division functions are:

- `rdivide`, `rdivide` and `wdivide` in the `CollateralAuctionHouse` contract.
- `rdivide` in the `OracleRelayer` contract.

To prevent such unsafe calculations, consider adding a require statement in the functions to ensure `y > 0`, or consider using the `div` functions provided in OpenZeppelin's `SafeMath` libraries.

Update: *Fixed in pull request #77.*

[M08] Not using SafeMath functions

There are several places in the code base where regular Solidity arithmetic operators are used. For example:

- In line 298 of the `GlobalSettlement` contract `/` is used.
- In line 196 of the `Coin` contract `++` is used.
- In line 565 of the `TaxCollector` contract `/` is used.

These operators do not protect against overflows, underflows or division by `0` and may silently fail or return unexpected values. Consider always performing arithmetic operations with functions that protect the code from such scenarios, like the math libraries of OpenZeppelin contracts.

Update: *Acknowledged, and will not fix. Reflexer Labs' statement for this issue:*

> We understand the concern although we would like to stick to the same functions used in MCD

[M09] `WAD` incorrectly used for rounding

Within `CollateralAuctionHouse`, if the amount being bid in a collateral auction exceeds the `remainingToRaise` amount, the bid will be set to the equivalent of `remainingToRaise`. Since `remainingToRaise` is in `RAD` form (scaled by `1e45`), it must be divided by `RAY` (`1e27`) to be in proper `WAD` form. This division results in truncation, where any result will be rounded down to the nearest integer value. However, since this value is the amount being paid into the system, it should be rounded up, to disallow users from paying less than they should. `WAD` is added after the division to counteract truncation losses, but `1` should be added.

Consider replacing `WAD` with `1` on line 806 of `CollateralAuctionHouse`.

Update: *Fixed in commit `02b2db5a85ed763deb436ad548e636c9efad1cde`.*

Low severity

[L01] Constant `HUNDRED` declared twice with different values

The constant HUNDRED is assigned the value `10**2` in StabilityFeeTreasury and the value `10**29` in TaxCollector.

To avoid confusion for future developers, consider re-naming one of the two instances to something else.

Update: *Fixed in pull request #78.*

[L02] `contractEnabled` should be a bool

The variable `contractEnabled` is declared in many contracts as a `uint` value, for example within the `BasicTokenAdapters` or the `SAFEEngine` contracts. Whenever its value is set, it is always set to a `1` or `0` value. Furthermore, whenever it is checked, it is always checked for strict equality to either `0` (like within the `AccountingEngine` contract) or `1` (also within the `AccountingEngine` contract).

Since a `uint` can contain many values besides `0` or `1`, if somehow any one of the `contractEnabled` values is set greater than `1`, all strict equality checks on `contractEnabled` will fail.

If this variable is intended to have only two possible values, consider changing the declarations of `contractEnabled` so that they are `bool`s. This is the intended type for such a situation, and will prevent accidentally setting `contractEnabled` to anything other than the two intended values.

Update: *Acknowledged, not fixed. Reflexer Labs' statement for this issue:*

> We would like to stick to the same logic inherited from MCD

[L03] Declare `uint` as `uint256` and `int` as `int256`

To favor explicitness, all instances of `uint` should be declared as `uint256` and all instances of `int` should be declared as `int256`.

Update: *Partially fixed in pull request #81. Two instances of non-explicit `uint` or `int` types were found on `AccountingEngine` line 134 (`uint rad`) and `CollateralAuctionHouse` line 607 (`uint(systemCoinPriceFeedValue)`).*

[L04] ERC20 token decimals should be of uint8 type

The ERC20 specification defines the token's `decimals` to be a `uint8` type. While the `Coin` token implements this correctly, for consistency, consider changing the type of the variable `decimals` in `BasicCollateralJoin`, `ETHJoin` and `CoinJoin` contracts from `uint` to `uint8`.

Update: *Acknlowledged, not fixed. Reflexer Labs' statement for this issue:*

> For now we would like to stick to `uint`

[L05] Lack of indexed parameters in events

In the codebase, almost all event definitions are lacking indexed parameters.

There are several contracts that, in some cases, correctly make use of indexed parameters, like the `Coin` contract that has two events using the `index` keyword, or `CollateralAuctionHouse`, which has indexed parameters in the `StartAuction` event, both in the `EnglishCollateralAuctionHouse` and in the `FixedDiscountCollateralAuctionHouse` contracts.

However, there are a few places in the codebase where indexed event parameters are used. Contracts like the `AccountingEngine` or the `SurplusAuctionHouse` completely lack them.

Consider reviewing all events in the codebase to index parameters. Indexed parameters assist off-chain services (like applications and user interfaces) in searching and filtering for specific events.

Update: *Fixed in pull request #82.*

[L06] Missing error messages in require statements

In the code base there are many places where `require` statements are correctly followed by their error messages, clarifying what was the triggered exception. However, there are places in the code where `require` statements are not followed by the corresponding error messages.

This happens frequently in the internal arithmetic functions such as the one in the `LiquidationEngine` or in the `StabilityFeeTreasury` contract. If any of those require statements fails the checked condition, the transaction will revert silently with no informative error message.

The mentioned examples are just a few of the total present in the code base.

Consider including specific and informative error messages in all `require` statements.

Update: *Fixed in pull request #83.*

[L07] Collateral is assumed to be 18 decimals

In the `BasicCollateralJoin` contract, it is assumed that the collateral token has 18 decimals, however there is no check in the adapter to ensure this.

To enforce this condition, consider adding a `require` statement in the constructor after line 103 which verifies that `decimals == 18`.

Update: *Fixed in pull request #84.*

[L08] Missing docstrings

Many of the contracts and functions in the Reflexer code base lack documentation. This hinders reviewers' understanding of the code's intention, which is fundamental to correctly assess not only security, but also correctness. Additionally, docstrings improve readability and ease maintenance. They should explicitly explain the purpose or intention of the functions, the scenarios under which they can fail, the roles allowed to call them, the values returned and the events emitted.

Functions which lack docstrings include:

- `CoinSavingsAccount.rpower`.
- `GlobalSettlement.calculateCashPrice`.
- `GlobalSettlement.fastTrackAuction`.
- `GlobalSettlement.freeCollateral`.
- `GlobalSettlement.freezeCollateralType`.
- `GlobalSettlement.prepareCoinsForRedeeming`.
- `GlobalSettlement.processSAFE`.
- `GlobalSettlement.redeemCollateral`.
- `GlobalSettlement.setOutstandingCoinSupply`.
- `GlobalSettlement.shudownSystem`.
- `TaxCollector.taxSingleOutcome` lacks explanation for its output parameters.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Acknowledged, not fixed. Reflexer Labs' statement for this issue:*

> We will have to delay this but we will take care of it.

[L09] Lack of input sanitization

Throughout the codebase, many contracts have an instance of the function `modifyParameters()`, which allows authorized accounts to change parameters of the protocol. For example, such functions exist in the `CoinSavingsAccount` contract and in the `GlobalSettlement` contract, among many others.

However, there are no safety checks for many of these parameters when they are being changed. For example, in the `GlobalSettlement` contract, no changes to `address` parameters check that the `address` given is not `address(0)`, or that there is a contract at that address. In the `AccountingEngine` contract, some "delay" parameters are not bounded, allowing them to be set to excessively high values. And in the `CollateralAuctionHouse` contract, some parameters are immediately cast to `uint48`, allowing undetected overflows if the value is higher than `2**48-1`.

Furthermore, values passed into `constructor`s for various contracts do not contain input checks. For example, the `constructor` for the `BasicCollateralJoin` contract contains no checks that the two `address` parameters are not `address(0)`, and neither does the `constructor` of the `CollateralAuctionHouse` contract.

Note that while these are examples, this is not an exhaustive list of all places needing input sanitization. To help follow the "fail early and loudly principle", ensure that for all instances of `modifyParameters` and for all `constructor`s, there are checks on the values being set. In general, consider checking that `address` parameters are not `address(0)`, that `address`es which correspond to contracts have `extcodesize != 0`, that `uint` variables which are then downcast have not overflowed, and that numeric parameters are bounded to reasonable values given their application.

Update: *Aknowledged, not fixed. Reflexer Labs' statement for this issue:*

> We understand the concern although we'd like to stick to what we inherited from MCD.

[L10] Repeated code

Throughout the code base, there is consistent use of repeated code. Some instances of repeated code are as follows:

- All the arithmetic functions, such as in `AccountingEngine` or in `OracleRelayer`, are repeated in many contracts. Note that these arithmetic functions are well covered by OpenZeppelin Math libraries, which are battle tested and widely adopted contracts.
- The governance functions `addAuthorization` and `removeAuthorization`, and the modifier `isAuthorized` are repetitive.
- Similarly, the `either` and `both` functions, which are just performing logical `OR` and `AND` operations, are often repeated in the code base.
- Lines 199-201 of the `disableContract` function in the `StabilityFeeTreasury` contract are the same as the `joinAllCoins` function.
- The conditions on lines 317-320 and line 348 of the `LiquidationEngine` contract are identical.
- Lines 489-491 and 353-355 of the `SAFEEngine` contract are repeated.
- Lines 717 and 726 of the `FixedDiscountCollateralAuctionHouse` contract are repeated.
- `getApproximateCollateralBought` and `getCollateralBought` functions of the `FixedDiscountCollateralAuctionHouse` contract are almost identical.

Solidity language provides the use of libraries to call functions that need to be accessed by several contracts and that are always the same. Having libraries is easy for code maintenance since any bug or new functionality can be coded in only one single contract that serves all the others. Also, libraries can also help in reducing the gas cost associated with deployment and use of the contracts while reducing the potential attack surface. Contracts can also define internal functions that are then called internally in many places.

Consider adopting libraries and internal functions design to improve code size, quality and readability at the same time.

Update: *Partially fixed in pull request #84. Lines 199-201 in `StabilityFeeTreasury` have been replaced with a call to `joinAllCoins`, and the `getAdjustedBid` function has been created to consolidate functionality for `getApproximateCollateralBought` and `getCollateralBought`.*

[L11] Uncommented assembly block

The `OracleRelayer` contract includes an assembly block in the `rpower()` function. The same assembly block is repeated in the `TaxCollector` and `CoinSavingsAccount` contracts.

Other functions like `either()` or `both()` in the `CollateralAuctionHouse`, `LiquidationEngine`, `SAFEEngine`, `StabilityFeeTreasury` and `TaxCollector` contracts are using assembly lines without any docstring or comment.

While this does not pose a security risk *per se*, it is at the same time a complicated and critical part of the system. Moreover, as this is a low-level language that is harder to parse by readers, consider including extensive documentation regarding the rationale behind its use, clearly explaining what every single assembly instruction does. This will make it easier for users to trust the code, for reviewers to verify it, and for developers to build on top of it or update it.

Note that the use of assembly discards several important safety features of Solidity, which may render the code unsafer and more error-prone. Hence, consider implementing thorough tests to cover all potential use cases of these functions to ensure they behave as expected.

Update: *Acknowledged, not fixed. Reflexer Labs' statement for this issue:*

> We will have to delay this.

[L12] Unnecessary require statements

There are several instances in the code base where the `require` statements or conditional checks are unnecessary. For instance:

- In the `OracleRelayer` contract, the `require` statement in the `modifyParameters` function at line 189 checks if the input parameter `data > 0`. This is unnecessary since the same condition is already checked in the `require` statement at line 187.

- In the `StabilityFeeTreasury` contract, the `require` statement in the `constructor` at line 113 checks if the input address `accountingEngine_` is not the same as `address(this)`. The scenario can happen only when the address of the `StabilityFeeTreasury` contract, that is going to be deployed, is precalculated and then passed to the constructor as the `accountingEngine_` address. Since the `StabilityFeeTreasury` contract is deployed by the governance, which is assumed to not be malicious, this check is unnecessary.

To simplify the code and prevent wastage of gas, consider removing the unnecessary checks.

Update: *Fixed in pull request #85.*

Notes & Additional Information

[N01] State variables and events declared after functions

Throughout the code base, variables and events are declared after functions. For example, in the `AccountingEngine` contract, the function `addAuthorization` uses the `contractEnabled` variable on line 55. This variable is declared further down in the contract on line 122.

To improve readability, consider declaring state variables, events and constructor before defining functions within a contract.

Update: *Acknowledged, not fixed. Reflexer Labs' statement for this issue:*

> For now we will keep the same structure.

[N02] Incorrect or misleading docstrings

In the code base there are several docstrings that are either incorrect or confusing. Examples are:

- Line 299 of the `AccountingEngine` contract says that the contract will automatically send any surplus right away. This is done by calling `transferPostSettlementSurplus` but this is not done within the function.
- Line 182 of the `CoinSavingsAccount` contract says "smaller" when it should be "greater than".
- Line 212 of the `LinkedList` contract should say "head or tail" rather than "head".
- Line 476 of `SAFEEngine` contract should say "debt" rather than "collateral".
- Line 85 of the `GlobalSettlement` contract says that `shutdownSystem()` will cancel collateral auctions. This is not true since it is done in the `fastTrackAuction()` function.

Consider reviewing all docstrings in the code base and fixing them to better reflect function behaviours and improve code readability.

Update: *Fixed in pull request #86.*

[N03] Catch clause not handled

In `getCollateralMedianPrice` and `getSystemCoinMarketPrice` functions of the `FixedDiscountCollateralAuctionHouse` contract, the `catch` clause of the `try/catch` is not emitting events nor handling the error, continuing the execution.

Even if continuing execution after a possible fail is something explicitly wanted, to follow the "fail early and loudly" principle, consider handling the `catch` clause by either emitting an appropriate event or registering the failed `try` call.

Update: *Acknowledged, not fixed. Reflexer Labs' statement for this issue:*

> We will keep them as they are right now.

[N04] Naming issues

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:

- Rename `SAFEEngine` struct `CollateralType` to `SAFECollateralInfo`.
- Rename `SAFEEngine` mapping `collateralTypes` to `SAFEInfoForCollateral`.
- Rename `OracleRelayer` struct `CollateralType` to `OracleCollateralInfo`.
- Rename `OracleRelayer` mapping `collateralTypes` to `oracleInfoForCollateral`.
- Rename `LiquidationEngine` struct `CollateralType` to `LiquidationCollateralInfo`.
- Rename `LiquidationEngine` mapping `collateralTypes` to `liquidationInfoForCollateral`.
- Rename `TaxCollector` struct `CollateralType` to `TaxCollateralInfo`.
- Rename `TaxCollector` mapping `collateralTypes` to `taxInfoForCollateral`.
- Rename `updateAccumulatedRate` function's parameter `rateMultiplier` to `rateIncrease`.

Update: *Acknowledged, not fixed. Reflexer Lab's statement for this issue:*

> We will keep the namings as they are right now.

[N05] Solidity compiler version is not pinned

Throughout the code base, consider pinning the version of the Solidity compiler to its latest stable version. This should help prevent introducing unexpected bugs due to incompatible future releases. To choose a specific version, developers should consider both the compiler's features needed by the project and the list of known bugs associated with each Solidity compiler version.

Update: *Fixed in pull request #80.*

[N06] `restartAuction` can be called for auctions which never started

In the `DebtAuctionHouse` contract, the function `restartAuction` can be called for auctions which have never started by passing in an `id` value greater than `auctionsStarted`.

This should have no meaningful effect, but consider implementing a check within `restartAuction` that enforces `id <= auctionsStarted` to improve user experience.

Update: *Fixed in pull request #87.*

[N07] Local variable can be reused

In line 869 of the `CollateralAuctionHouse` contract, the call to the `subtract()` function can be replaced by the `leftoverCollateral` variable defined two lines above.

In order to save gas and improve code understandability, consider replacing the `subtract()` call with the `leftoverCollateral` variable.

Update: *Fixed in commit `c582fb57c746e36ed6f43ca80a7816e751c0ae2d`.*

[N08] Strict equality to `now` exists

Within the code base, there are some places where a `require` compares some value to `now`, the block timestamp. When these `require` checks fail, calls will be reverted. Some examples of this are:

- Within the `CoinSavingsAccount` contract, on line 155 and on line 210, `latestUpdateTime` is compared to `now`. `updateAccumulatedRate` will set `latestUpdateTime` appropriately, and will return early if `latestUpdateTime` has already been set for that block.

- Within the `TaxCollector` contract, on line 241, `collateralTypes[collateralType].updateTime` is compared to `now`. This value is updated within `taxSingle`, and `taxSingle` will return early if it has already been called for that `collateralType` within the same block.

- Within the `OracleRelayer` contract, on line 193, `redemptionPriceUpdateTime` is compared to `now`. This value is set within `updateRedemptionPrice`, which is called by `redemptionPrice`. It will return early if called more than once within the same block.

Consider replacing the identified `require` checks with calls to the functions which set the values they are checking instead. Replacing these `require`s will greatly improve user experience by merging two potential transactions into one, saving on gas costs and reducing the chances of a reverted transaction. Since these requires are at the beginning of `external` or `public` functions, and the functions which would replace them are also `external` or `public`, these functions can already be called back-to-back, and replacing the requires with the proper function calls will behave the same as calling them back-to-back.

Update: *Partially fixed in pull request #88 and pull request #86. Only the identified instances within `CoinSavingsAccount` were fixed. Reflexer Labs' statement for this issue:*

> We will keep them as they are right now apart from the CoinSavingsAccount.

[N09] Typos

There are several typos in the code base.

- On line 116 of the `AccountingEngine` contract, "surpluscan" should be "surplus can".
- On line 131 of the `BasicTokenAdapter` contract, "adapte" should be "adapter".
- On line 164 of the `BasicTokenAdapter` contract, "an authed" should be "a restricted".
- On line 803 of the `CollateralAuctionHouse` contract, "it's" should be "is".
- On line 331 of the `GlobalSettlement` contract, "art" should be "safeDebt".
- On line 10 of the `LinkedList` contract, "an utility" should be "a utility".
- On line 128 of the `LinkedList` contract, "_value" should be "StructLike(_struct).val(next_)".

- On line 212 of the `LinkedList` contract, "head" should be "head or tail".
- On line 129 of the `OracleRelayer` contract, "alsites" should probably be "always".
- On lines 162 and 218 of the `OracleRelayer` contract, "who's" should be "whose".
- On lines 296, 303 and 310 of the `OracleRelayer` contract, "collateral price" should be "collateral type".
- On line 61 of the `SafeEngine` contract, "give" should be "deny".

Consider fixing them to improve readability and overall quality of the code base.

Update: *Fixed in pull request #88.*

[N10] Unclear variable role

In the `LiquidationEngine` contract in lines 23 and 371, the `initialBidder` variable is used to represent what in the `CollateralAuctionHouse` contracts is called `auctionIncomeRecipient`.

The `CollateralAuctionHouse` contracts are actually using the passed address as the one receiving system coins obtained by collateral auctions and for this, the `initialBidder` variables in the `LiquidationEngine` contract should be renamed to better reflect that they are the recipient of system coins and not the first bidders.

Update: *Acknowledged, and will not fix. Reflexer Lab's statement for this issue:*

> Same as M05, Maker made the interface general in the sense that we could have a future implementation where someone triggers an auction and also places the first bid.

[N11] Unnecessary event emission

The `popDebtFromQueue` function of the `AccountingEngine` contract is emitting a useless event whenever someone tries to call it with a `debtBlockTimestamp` that has not been saved before.

Consider checking if `debtQueue[debtBlockTimestamp]` is greater than 0 before anything else to save gas and avoid emitting unnecessary events.

Update: *Fixed in pull request #89.*

[N12] `rmultiply()` is not used in LiquidationEngine contract

In the `LiquidationEngine` contract, the `internal` function `rmultiply()` is unused in the audited codebase.

Consider removing it to simplify the code and improve readability.

Update: *Fixed in pull request #90.*

[N13] Unused variables

In the `FixedDiscountCollateralAuctionHouse` contract, inside the `getApproximateCollateralBought` and `getCollateralBought` functions, the `totalRaised` local variable is declared but not used anywhere else.

In the auction-based contracts, such as `EnglishCollateralAuctionHouse`, `FixedDiscountCollateralAuctionHouse`, `DebtAuctionHouse`, `PreSettlementSurplusAuctionHouse` and `PostSettlementSurplusAuctionHouse`, the constants `AUCTION_HOUSE_TYPE` and `AUCTION_TYPE` are declared but are never used in the code.

Lastly, the `RAD` constant in the `FixedDiscountCollateralAuctionHouse` is declared but never used.

Consider removing the `totalRaised` variable and any unused constant from the code base.

Update: *Partially fixed in pull request #91. Instances of `AUCTION_HOUSE_TYPE` and `AUCTION_TYPE` have been intentionally left in the codebase. Reflexer Labs' statement for this issue:*

> We want to leave these in because they're used by other contracts and also by keepers to differentiate between auction types

Conclusions

No critical and one high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.