Money on Chain

July 2019

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the Money On Chain project. We will provide an executive summary of our discoveries, a short description of the project, the methodology used, the details of our findings and will finish with our conclusion of the code audited.
There are two appendices that include the listing of contracts audited and our analysis of the more important public functions of the contracts.

Executive Summary

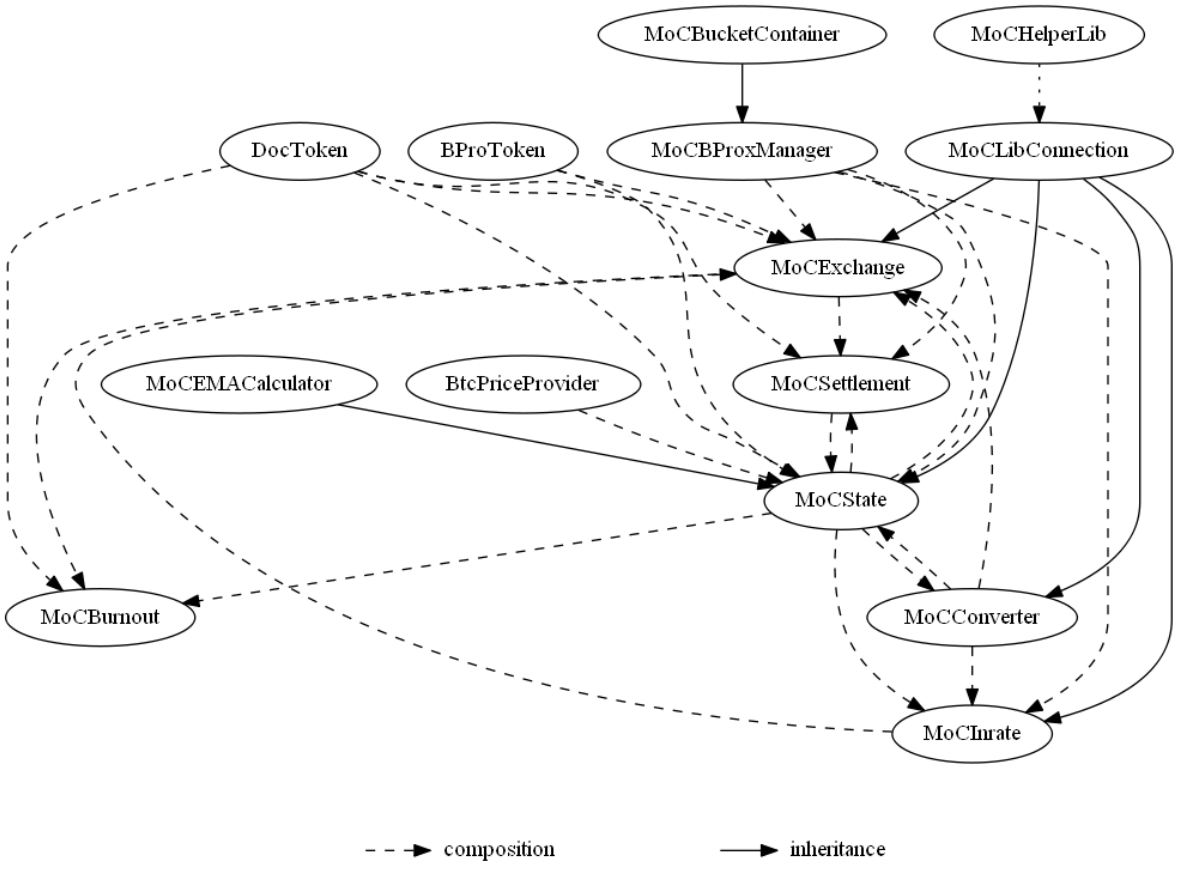
This is the second audit we perform on the code of the Money On Chain project.
We didn't found issues with critical or high risk.
We found one issue with medium risk since it involves the contracts initialization. From our tests it cannot be exploited. We notified the team so they document it properly or make proper fixes.
There are a few minor issues/enhancements that do not affect the contract functionality, for example contracts variable with helpful names, or some error conditions which will not generate a message.

Contracts

The contracts audited are from the Money on Chain project. The audited contracts consist of 3 distinct functionalities:

- Money On Chain: Money On Chain is a suite of smart contracts dedicated to providing a bitcoin-collateralized stable-coin.
- Governance: A suite of smart contracts dedicated to providing a governance system which is generic enough to work without knowing the system to be governed.
- Oracle: Money on Chain USD-BTC price provider.

The following dependency graph shows the most important contracts from the Money On Chain repository:



Note: The dashed lines indicate composition and the solid lines indicate inheritance.

Analyses performed

The following analyses were performed:

- Misuse of the different call methods: call.value(), send() and transfer().
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions.
- Old compiler version pragmas.
- Race conditions such as reentrancy attacks or front running.
- Misuse of block timestamps, assuming anything other than them being strictly increasing.
- Contract softlocking attacks (DoS).
- Potential gas cost of functions being over the gas limit.
- Missing function qualifiers and their misuse.
- Fallback functions with a higher gas cost than the one that a transfer or send call allows.
- Fraudulent or erroneous code.
- Code and contract interaction complexity.
- Wrong or missing error handling.

- Potential overuse of transfers in a transaction might produce unnecessary fees. Withdrawal pattern should be used instead.
- Insufficient analysis of the function input requirements.

Detailed findings

Medium severity

Usage of array deletes may surpass gas limits

Array deletes have a linear cost even when they reimburse gas. Gas reimburse is applied after the gas limit checks are done. Therefore even if the final gas calculation lies below the gas limit the transaction may still fail if it surpassed the gas limit at some point.
For this reason a delete operation can fail due to gas costs if the array is big enough leading to DoS. There are two array deletes. One in *MoCSettlement*:

```
function clear() public onlyWhitelisted(msg.sender) {
    delete redeemQueue;
}
```

and one in *MoCBucketContainer*:

```
function clearBucketBalances(bytes32 bucketName) public
onlyWhitelisted(msg.sender) {
    MoCBucket storage bucket = mocBuckets[bucketName];
    bucket.nBPro = 0;
    delete bucket.activeBalances;
}
```

We recommend revising the gas cost of these operations. You may need to add paging to these operations.

Minor severity

Conflicts on secure contract initialization

The project uses ZeppelinOS to allow upgrades without having to migrate contract's data. To achieve this it has inherit from *Initializable* from ZeppelinOS which provides a modifier *initializer* for secure initialization of upgradeable contracts.

But it also inherits from *MoCBase*, which provides a similar modifier *onInitialization* to protect the contract against unsecure initialization.
This is an issue, *Initializable* assumes the first function to be called will be modified by *initializer* to ensure that only that function will set *initialized* to *true* after returning. But this is not what happens, as the first function is *onInitialization* which later initializes *Stoppable* which is *Initializable*. if another contract using *Initializable* was inherited and initialized by this function, it will fail as Stoppable already set *initialized* to *true*.
We recommend only using the ZeppelinOS version to avoid these types of issues in the future.

Storage variable with the same name

The MoC contract inherits from *Initializable* and *MoCBase*. They use a variable with the same name *initialized*, it is private in ZeppelinOS but it is internal in MoCBase which allows access from derived contracts. Since solidity allows multiple inheritance and uses C3 linearization to determine the precedence order it is possible that a change in the inheritance order in a derived contract will affect the variable being referenced.
We suggest to always declare variables with the least possible scope, since *initialized* is not used outside of MoCBase is better to declare it as private.

Old solidity version

The contracts for the Oracle project require solidity version 0.4.24 which was released on May 2018. While we didn't find any vulnerability related to using this specific version, we recommend upgrading to a more recent version as many issues and ambiguities get fixed in each release. If contracts can't be upgraded to v0.5 you should consider using v0.4.26 the latest version of the v0.4 branch.
For example in *oracle\contracts\price-feed\price-feed.sol* we have:

```
pragma solidity ^0.4.23;
```

Unhelpful variables names

There are a few instances in the Oracle contracts that variables and parameters have names that are not helpful to understand the code.
For example function *read()* in *lib/value.sol*

```
function read() public view returns (bytes32) {
    bytes32 wut; bool haz;
    (wut, haz) = peek();
    require(haz, "haz-not");
    return wut;
}
```

Missing message in requires

The *require()* statement has an optional message parameter that will return in case of failure of the testing condition.

- oracle/contracts/lib/math.sol: the function *add()*, *sub()* and *mul()* have a require without message

```
function add(uint x, uint y) internal pure returns (uint z) {
    require((z = x + y) >= x);
}
```

Observations/Remarks

Upgradability

A major change introduced to Money On Chain in the second audit was to make contracts upgradeable using ZeppelinOS.
One important benefit of this feature is that in case of bugs the contracts can be upgraded without requiring changes to third party tools or intervention from users. Also contracts data doesn't have to be migrated lowering costs.
This feature has the drawback that there is one special account that control the upgrade. It is possible for the entity controlling this account to upgrade the contracts to a completely different version without approval from users.
There is also a new feature in the contracts called "changers" that have similar but reduced implications. It allows the owner to grant permission to an arbitrary contract, the changer, for a single transaction to make changes to storage variables inside the project. That is, it allows the owner to modify multiple selected variables on different contracts in a single transaction, instead of multiple ones which may cause issues, by delegating the task to another deployed contract. Since the whitelisting is decided by the owner at the moment of the transaction it can be used to execute other changers that are not included in this audit. This is not considered a vulnerability since the variables that changers are allowed to mutate are pre-selected and it's similar to having other privileged functions that can alter parameters. It does however, increase the surface area for errors if not handled with care.

Conclusion

We consider the contracts to be well written and abundantly documented, they use reasonable recent version of popular frameworks like OpenZeppelin, ZeppelinOS and most code use solidity compiler version 0.5.

We found a medium severity issue that is not exploitable regarding contracts initialization. A few minor issues that do not affect functionality and are about code style, using an old compiler version.
We also add the observation that upgradability can be considered a feature but it can also be considered a bug because it allows the owner to arbitrarily change the deployed bytecode.

Appendix: Audited files

The audited contracts grouped by project are:

Repository	Contract	Previous Audit
Governance	./Stopper/Stoppable.sol	No
	./Upgradeability/UpgradeDelegator.sol	No
	./ChangersTemplates/UpgraderTemplate.sol	No
	./Governance/Governor.sol	No
	./Governance/Governed.sol	No
	./Stopper/Stopper.sol	No
	./Governance/IGovernor.sol	No
	./Governance/ChangeContract.sol	No
Oracle	./medianizer/medianizer.sol	No
	./lib/math.sol	No
	./lib/auth.sol	No
	./price-feed/price-feed.sol	No
	./lib/value.sol	No
	./lib/note.sol	No
	./authority/MoCGovernedAuthority.sol	No
	./price-feed/feed-factory.sol	No
	./lib/thing.sol	No
	./MocMedianizer.sol	No
MoC	./MoCInrate.sol	Yes
	./MoCState.sol	Yes
	./base/PartialExecution.sol	No
	./MoC.sol	Yes
	./MoCHelperLib.sol	Yes
	./MoCSettlement.sol	Yes
	./changers/MocInrateChanger.sol	No
	./MoCEMACalculator.sol	No
	./changers/MocStateChanger.sol	No
	./changers/MocChanger.sol	No

	./MoCBucketContainer.sol	Yes
	./changers/MoCBucketContainerChanger.sol	No
	./changers/productive/PriceFeederAdder.sol	No
	./MoCExchange.sol	Yes
	./MoCBurnout.sol	Yes
	./changers/MoCSettlementChanger.sol	No
	./changers/productive/PriceFeederRemover.sol	No
	./changers/MoCStallSettlementChanger.sol	No
	./changers/MoCRestartSettlementChanger.sol	No
	./MoCBProxManager.sol	Yes
	./MoCLibConnection.sol	Yes
	./base/MoCConstants.sol	No
	./interface/BtcPriceFeed.sol	No
	./base/MoCBase.sol	Yes
	./interface/BtcPriceProvider.sol	No
	./MoCConverter.sol	Yes
	./base/MoCWhitelist.sol	Yes
	./base/MoCConnector.sol	Yes
	./token/OwnerBurnableToken.sol	Yes
	./test-contracts/RevertingOnSend.sol	Yes
	./token/BProToken.sol	Yes
	./token/DocToken.sol	Yes

Appendix: Function Analysis

The following graphs show the flow of a set of important functions in the Money On Chain project. Payable functions are the ones that send RBTC and Redeem functions are used to withdraw RBTC from the contract, so they are the main interaction point between the users and the project. These functions tend to be the most sensible and complex ones since they need to handle the currency. As such, they are the most prone to be vulnerable to attack vectors.
Since these functions span multiple contracts, which in turn implies a call stack per contract, a graph is useful to show how many of these contracts are reached and what specific functions they call for each case. The following graphs are meant to do that, specifying which contract, functions and point of entry may be reached. Note that this doesn't necessarily happen in a single call, as some functions may not get called depending on the state of the contracts and the input given.

How to read the graphs

The graphs are coded to ease the analysis:

- Internal calls are represented by black arrows. Internal calls do not make a new call stack since they are made inside the same contract.
- External calls are represented by red arrows. External calls do create a new call stack entry, as they need to call a contract on a different address.
- Functions are represented by black boxes ▭, and these are grouped into individually deployed contracts represented by the blue boxes ▭.
- Modifiers are represented by house shape ⌂, these are solidity constructs that decorate functions to provide functionality that is executed before and/or after the function that's being decorated.
- A dashed border implies the graph expands further from that node but was removed to simplify the graph.

We include SVG version of the graphs in a separate attachment.

Fallback function

The fallback function simply adds more balance (RBTCs) to the system. It also updates the bucket and the global variable which track this balance. It doesn't create any DoC token, BPro token or BProx instrument. Being simple, it doesn't consume much gas.

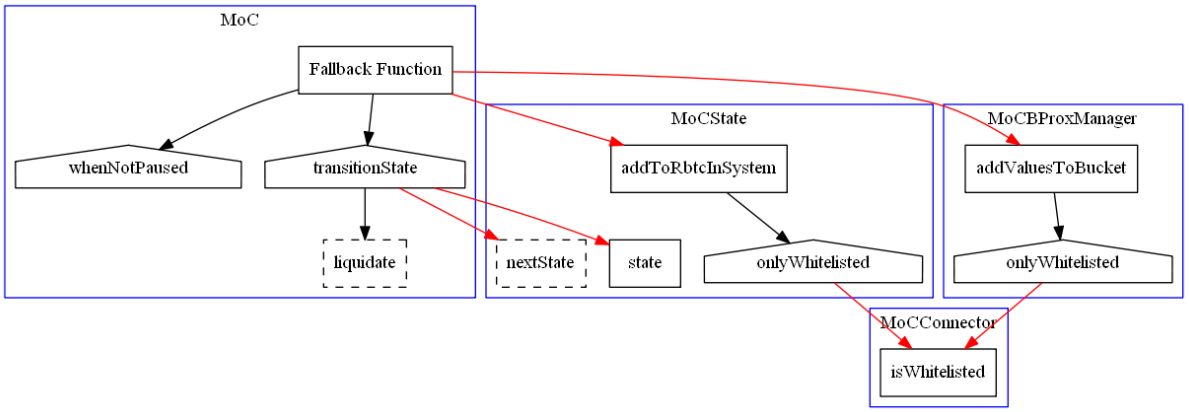


Figure 1: Fallback function.

The mintBPro function

This function creates BProTokens in exchange for RBTC and assigns them to the user.
If a discount rate is applicable (Which happens when the *BProDiscount* state is set) a limited amount of tokens is bought at a discounted price, and the rest of the RBTC is used to buy tokens at a normal rate.
The contract MoCConverter is used as a bridge that combines the values saved in MoCState and the functions in MoCHelperLib which contain the formulas. This allows the contract to provide easy access to conversion functions, which are needed to calculate the discount price. Even though there are many interactions between the contracts, most of it is retrieving values that are needed for the calculations, plus making the calculations themselves which reside in separate contracts.
The BProToken is obviously called, to mint the corresponding tokens that were bought, only the MoCExchange is able to mint tokens as it is the owner of the token contract.
As with the fallback function, it also updates the bucket and the global variables which track both the RBTC given to the contract and the tokens bought.
Since no part of this function varies with the input or the state, the gas consumption should not vary greatly.

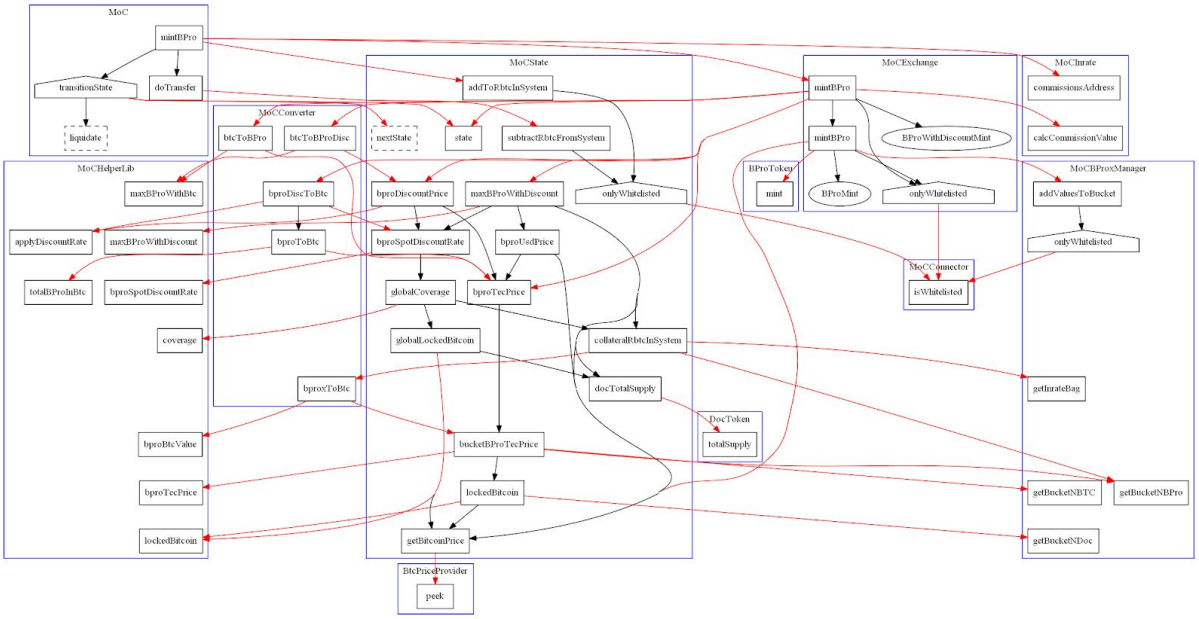


Figure 2: mintBPro function.

The mintBProx function

This function creates BProx in exchange of RBTC and assign them to the user.
The function *mintBProx* in MoC contract works as a frontend for MoCExchange *mintBProx* which coordinates the calls to other contracts. It calculates the maximum amount of BProx that can be calculated while maintaining the peg of the system and the amount of interest to pay in advance for the allocated BProx instruments.
For the calculations the current state is consulted from several contracts MoCState (bitcoin price, leverage level), MoCInrate (interest to pay), DocToken (token supply), MoCSettlement (next settlement block), MoCBProxManager (amount of Doc, BPro and RBTC in the bucket) and BtcPriceProvider (bitcoin price).
Once the amount of BProx and the interests are determined the allocation of BProx and buckets updating is done in the function *assignBProx* and *moveBtcAndDocs* from MoCBProxManager.
Other contracts like MoCInrate, MoCConverter, MoCHelperLib provide helper functions with calculations for intermediate values.
Since no part of the function varies with the input or the state, the gas consumption should be flat. The maximum cost will happen when an inexistent user in the system mint BProx, this is because this task requires allocating storage. But in general the contract mainly does calculations, so gas cost should not be high.

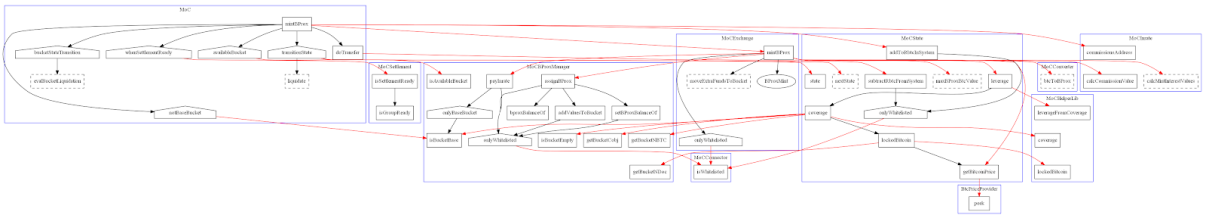


Figure 3: mintBProx function.

The mintDoc function

This function creates new DoC tokens in exchange of RBTC and assign them to the user.
The MoC contract will execute the function mintDoc in MoCExchange. It calculates the maximum amount of Doc tokens allowed to create while it maintains the peg. If the amount of RBTC sent is more than the amount needed to create new tokens it will be returned to the user at the end.
The new tokens are assigned to the user calling mint from DocToken contract. The minted Doc tokens are added to the bucket C0 in BProxManager contract.
Most of the functionality used in this call came from the MoCState contract to consult the state of the system to determine the maximum amount of Doc token available to the user. MoCHelperLib and MoCConverter contract provide helper functions to intermediate calculations.
No large variations should be expected in gas cost, being the maximum when a new user creates tokens because it involves allocating unused storage slots.

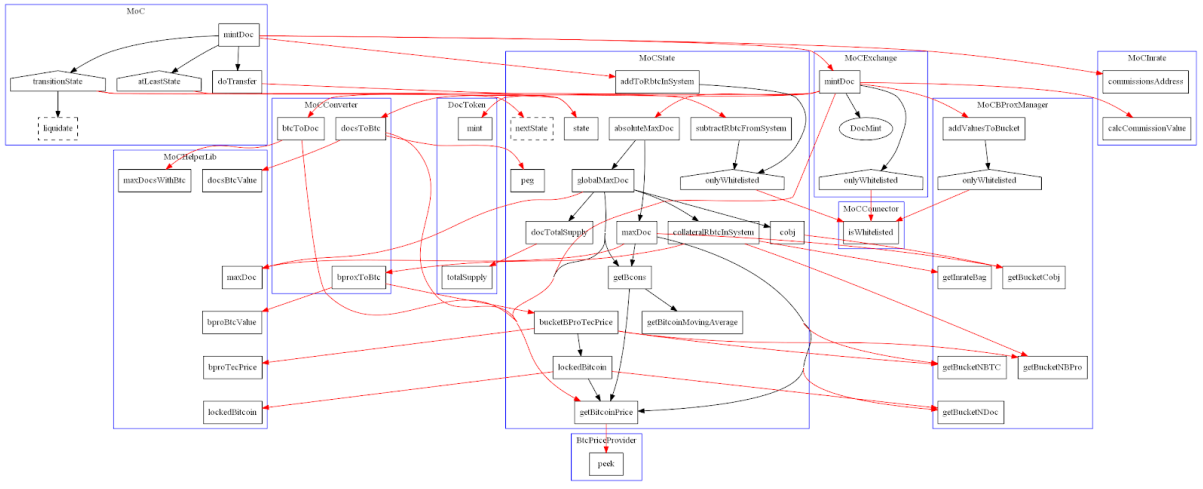


Figure 4: mintDoc function.

The redeemBPro function

This function redeems BPro token for RBTC. In order to redeem BPro tokens the system has to be in a healthy state with enough coverage.
The BPro tokens are burned and the RBTC is send to the user. A percentage of the RBTC is discounted as commission and sent a special address.
The MoC contract uses MoCExchange to convert the BPro amount into RBTC and the commission value. MoCExchange calculates the equivalent price of the BPro tokens in RBTC from the state in the contract, making sure it respects the maximum amount of BPro allowed and user balance. MoCExchange will burn the BPro tokens, update the bucket state in MoCBProxManager and generate events signaling the final values.
Once the tokens are burned MoC will send the amount of RBTC calculated to the user and the commission to a separate address.
No large variations should be expected in gas cost since all the operations done by the function have bounded gas use.

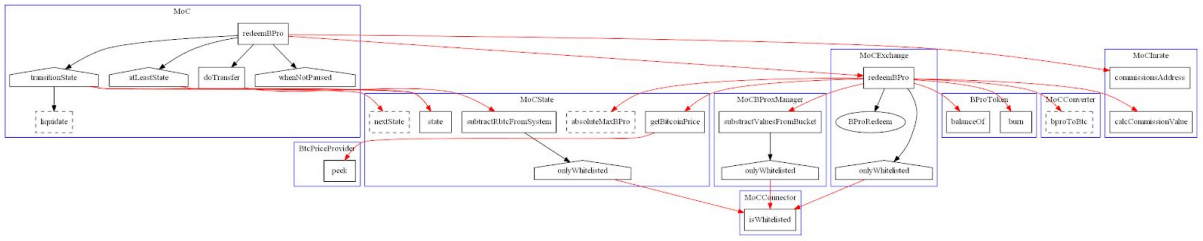


Figure 5: redeemBPro function.

The redeemBProx function

This function redeem BProx tokens for RBTC.
The MoC contract invokes redeemBProx function in MoCExchange. The function in MoCExchange will compute the interest that will receive the user and the commission it will pay. It will also burn the BProx tokens. It updates the bucket that contained the redeemed BProx tokens.
The MoC contract will transfer the RBTC to the user and send the commission to the proper address.

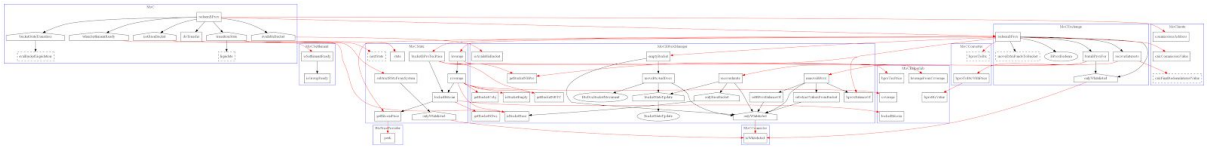


Figure 6: redeemBProx function.

The redeemFreeDoc function

This function allows a user to redeem a selected amount of free Doc Tokens for RBTC. In particular, free Doc Tokens are those found in bucket C0, if none are available this function will not be able to redeem tokens. The tokens are burned and the resulting RBTC are sent to the user. The function both collects an interest and a commission from the aforementioned RBTC that is saved in the bucket and sent to a commission address respectively.
To achieve that, it calls another function in MocExchange called redeemFreeDoc. This function will return the amount to RBTC that need to be sent to the user and the commission address respectively. In order to do so, it needs:

- To calculate how many tokens it will actually redeem and burn, since the requested tokens are capped by both the balance and the amount of free doc tokens. It does this by consulting both the bucket via MocState and the balance of the user via the token contract.
- To convert the Doc value to RBTC, it does this by calling the MoCConverter contract which will eventually go through MoCState and then the Oracle to get the RBTC price in USD.
- To calculate and save the interest that it will subtract from the previous RBTC conversion, it does this by calling the MocInrate contract.
- To calculate the commission that will subtract from the RBTC sent, it also does this by calling the MocInrate contract.
- Actually burn the tokens, both from the balance and the bucket and return the RBTC values accordingly.

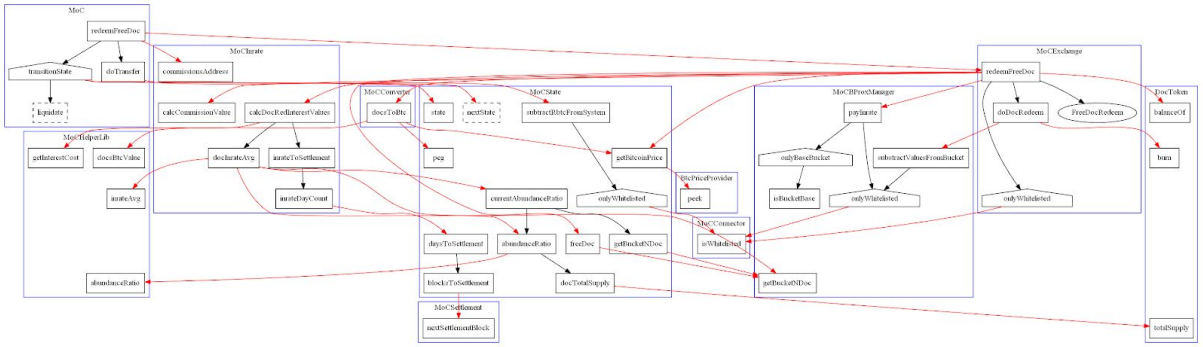


Figure 7: redeemFreeDoc function.