Money on Chain

April 2019

By Coinfabrik

Introduction

Coinfabrik was asked to audit the contracts for the Money On Chain project. Firstly, we will provide a summary of our discoveries and secondly, we will show the details of our findings.

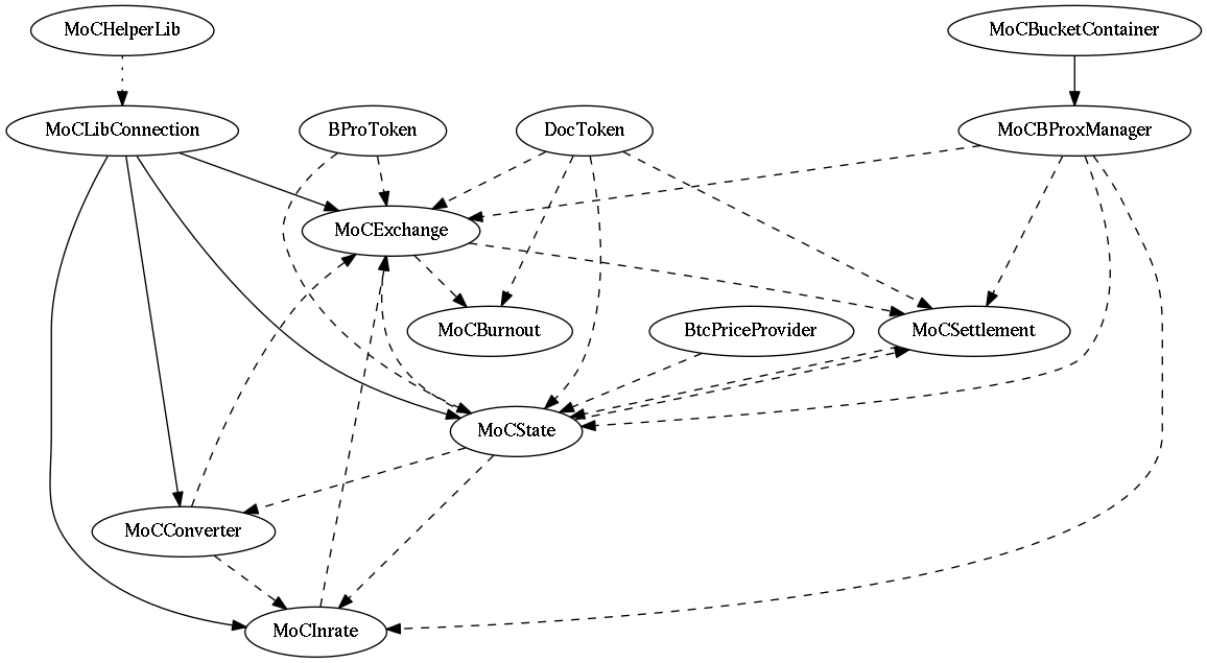
Summary

The contracts audited are from the Money on Chain project.

The audited contracts are:

- BtcPriceProvider.sol: An oracle that provides the price of BTC. Currently it's a mock.
- MoCBProxManager.sol: Manipulation of BProx balances.
- MoCBucketContainer.sol: Storage of BProx balances.
- MoCBurnout.sol: Burnout queue in case of liquidation.
- MoCConverter.sol: Exchange conversion between used units.
- MoCExchange.sol: Core exchange functionality (Minting/Redeeming)
- MoCHelperLib.sol: Helper functions used by other contracts.
- MoCInrate.sol: Calculates interest rates.
- MoCLibConnection.sol: Constants used in arithmetic operations.
- MoCSettlement.sol: Manages the settlement process.
- MoC.sol: Project main contract entry point.
- MoCState.sol: The core state of the project.
- base/MoCBase.sol: Base contract for all contracts, provides access control and initialization.
- base/MoCConnector.sol: Access control for all contracts using the whitelist.
- base/MoCWhitelist.sol: Simple whitelist implementation
- token/OwnerBurnableToken.sol: Allows the contract Owner to burn user tokens.
- token/BProToken.sol: ERC20 Mintable Token.
- token/DocToken.sol: Owner-Burnable Token, the stable coin.

You can see the dependency graph of the most important contracts below. Note that dashed lines indicate composition, while non dashed lines indicate inheritance:



Analyses 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.
- Overuse of transfers in a single transaction instead of using withdrawal patterns.
- Insufficient analysis of the function input requirements.

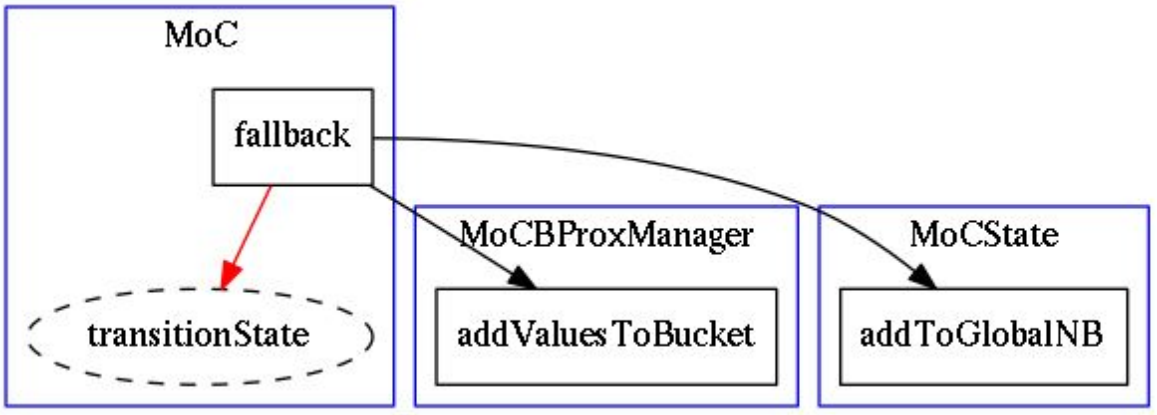
Analysis of payable functions

The following graphs show the flow of all payable functions in the project. Payable functions are the ones that send RBTC to 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 sent. 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

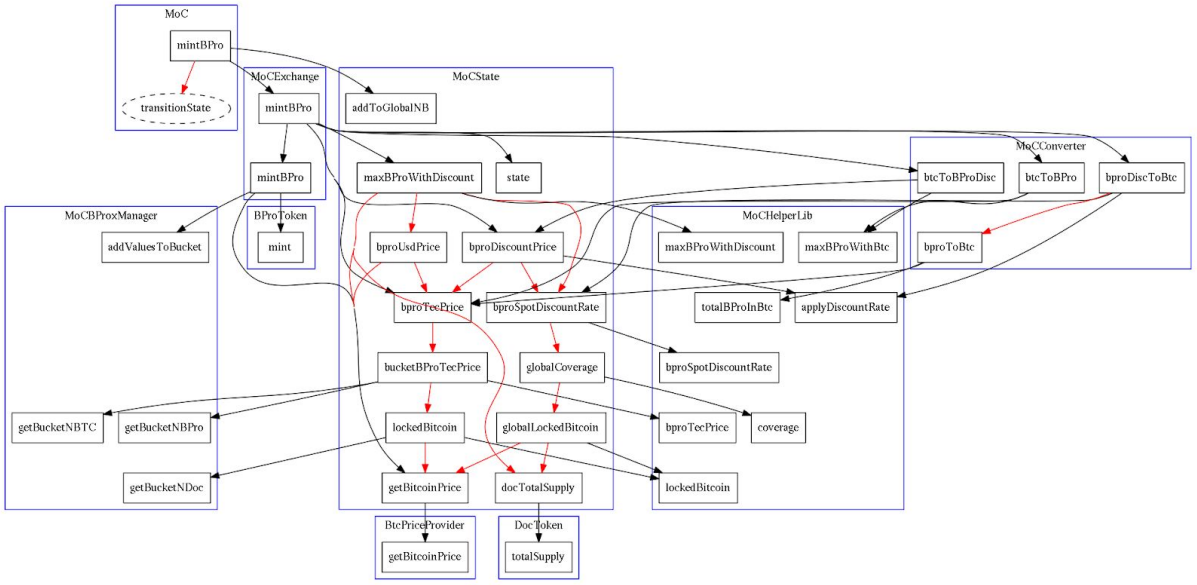
The graphs are color coded to ease the analysis.
Internal calls are represented by red arrows, while external calls are represented by black arrows. Internal calls do not make a new call stack since they are made inside the same contract. External calls do so, as they need to call a contract on a completely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

Fallback function



The fallback function simply adds more balance (RBTCs) to the main contract. 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.

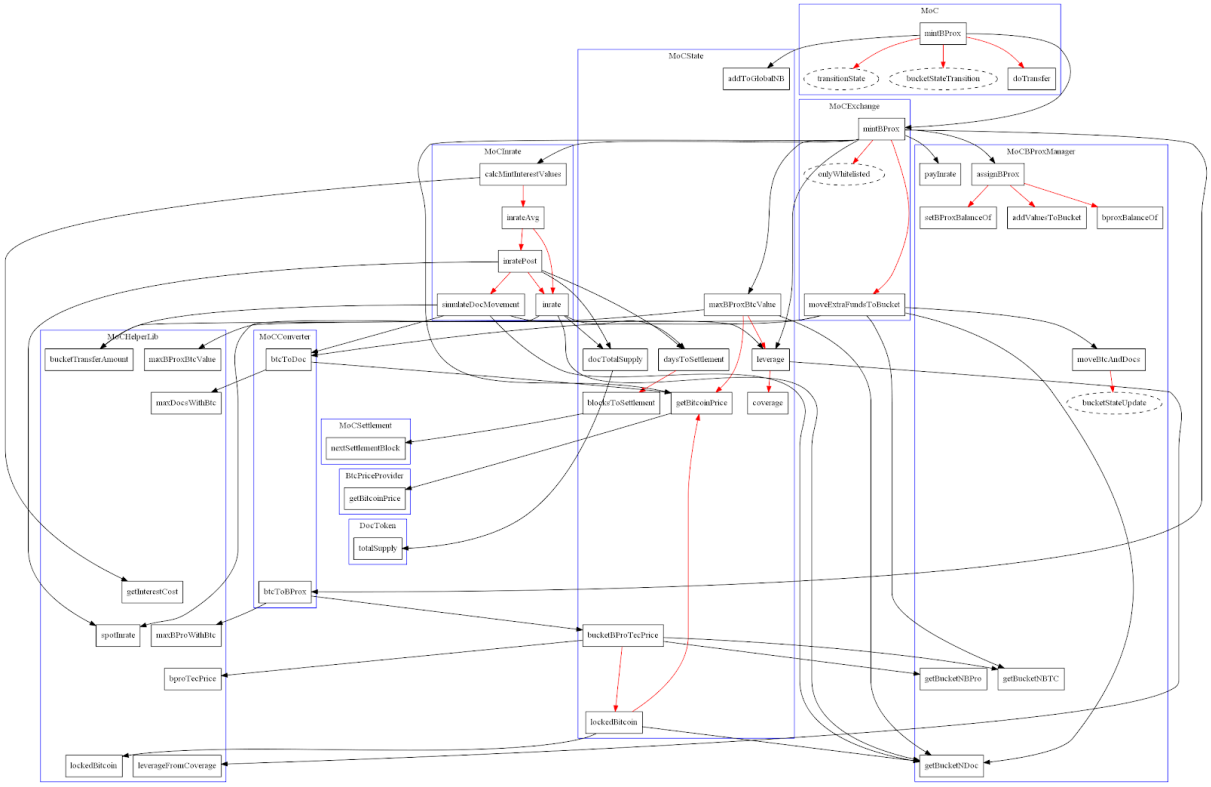
The mintBPro function



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.

The mintBProx function

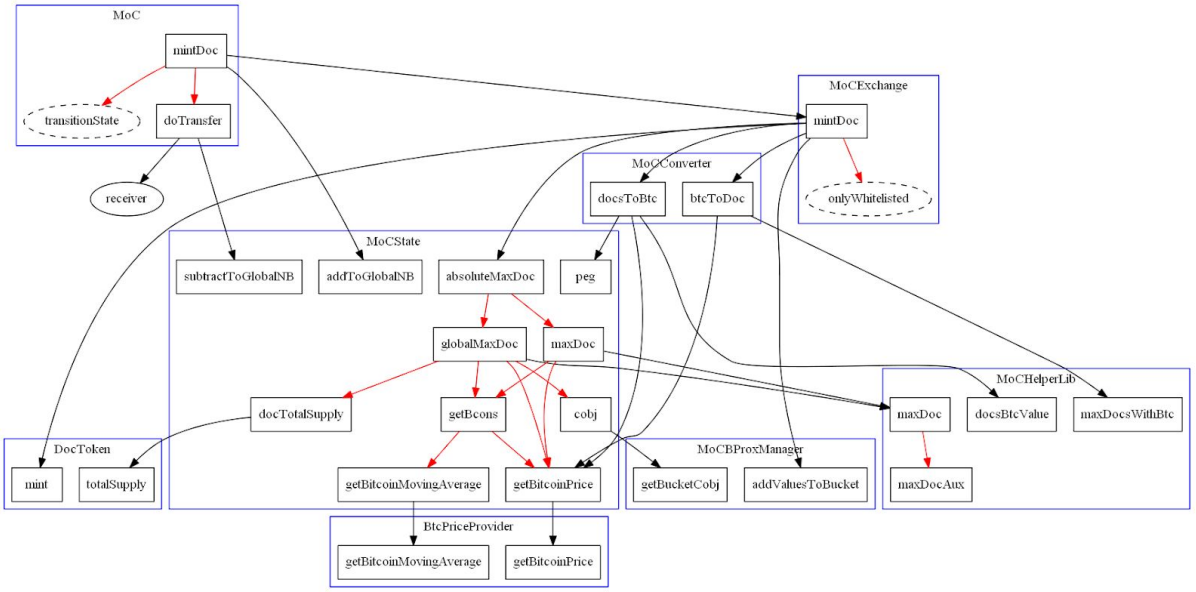


This function creates BProx in exchange of RBTC and assign them to the user.

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.

The mintDoc function

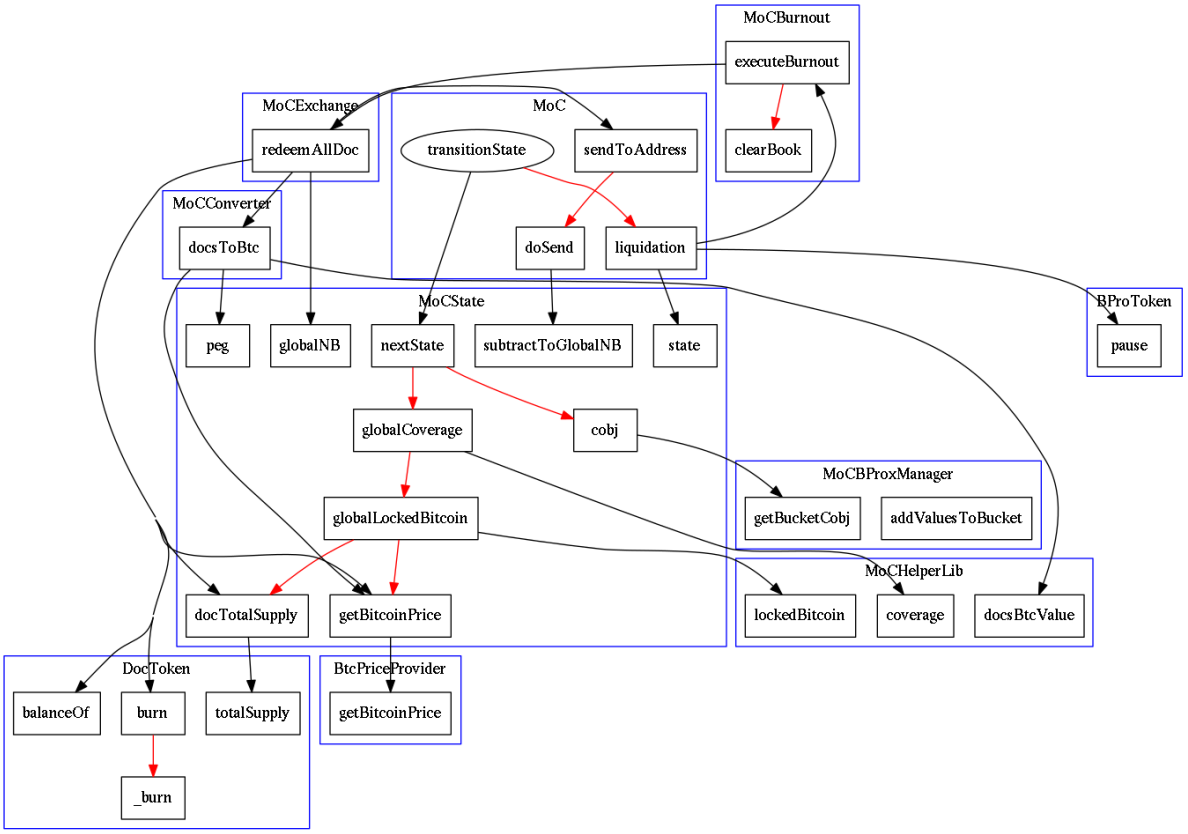


This function creates new DoC tokens in exchange of RBTC and assign them to the user.

The MoC frontend 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 "C0" bucket 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.

The transitionState modifier



The transitionState modifier is executed at the beginning of each externally callable function in the project. It checks in which numeric range the global coverage lands, and sets the associated state in the project. Depending on which state the project is, different actions will be taken when these functions are called. Specific states are required to call some functions, **redeemBPro** and **mintDoc** both require the state *AboveCobj* to be set, the **redeemAllDoc** function requires the *Liquidated* state. Once the *Liquidated* state is reached, which is the lowest numeric range (In other words, if the coverage becomes low enough), the contracts cannot recover from it and the project stays liquidated forever. When liquidation is reached, the burnout contract is called and all the users queued for burnout will get a refund of their Doc tokens. The price is either the current set price in RBTC or the contract balance divided by the total supply, whichever is lower.
Since the liquidation phase requires iterating through an entire queue, the gas consumption of this modifier at liquidation is unbounded.

The OnlyWhitelisted modifier

Also note that these graphs do not include the modifier OnlyWhitelisted, which is present in every external function that should be called only within the project contracts, and not by users or other contracts. This modifier checks if the calling address is one of the contracts within the project, and reverts otherwise. This ensures that no external actor is able to interact with the project internals, which stops a malicious actor from attacking the contracts. This modifier does not consume much gas, as it's only checking if an address is included in a set of addresses.

Detailed findings

Medium severity

An address can only be liquidated once and it may fail

The function **executeBurnout** liquidates all addresses residing in a queue:

```
/**
  @dev Iterate over the burnout address book and redeem docs
  for all users sending the RBTC to fhe corresponding burnout address
*/
function executeBurnout() public onlyWhitelisted(msg.sender) {
  for (uint i = 0; i < numElements; i++) {
      address account = burnoutQueue[i];
      address burnout = burnoutBook[account];
      uint256 btcTotal = mocExchange.redeemAllDoc(account, burnout);
      emit BurnoutAddressProcessed(account, burnout, btcTotal);
  }
  emit BurnoutExecuted(numElements);
  clearBook();
}
```

Users may add themselves into this queue by calling the function **pushBurnoutAddress** through **setBurnoutAddress** in MoC:

```
 /**
   @dev push a new burnout address to the queue for _who
   @param _who address for which set the burnout address
   @param _burnout address to send docs in liquidation event
 */
```

```
function pushBurnoutAddress(address _who, address _burnout) public
onlyWhitelisted(msg.sender) {
  require(_burnout != address(0x0), "Burnout address can't be 0x0");

  if (burnoutBook[_who] == address(0x0)) {
      pushAddressToQueue(_who);
  }
  burnoutBook[_who] = _burnout;
  emit BurnoutAddressSet(_who, _burnout);
}
```

The redeem may fail for a single user if they are interacting using a contract like a multisignature wallet. The problem is that **pushBurnoutAddress** does not allow a user to push itself into the queue twice, even if the redeem failed the first time. If that's the case, the user will be removed from the queue and will not be able to add itself into it again, being unable to redeem the tokens using that address. The only workaround available for the user is to move the tokens to a different address and try again.

We recommend resetting the **burnoutBook** address to zero for all users that were iterated in the **executeBurnout** function, which will allow users to be added into the queue again.

Multiple unbounded loops may result in Denial of Service

Multiple loops found in contracts MoCBucketContainer, MoCBurnout.sol and MoCSettlement.sol iterate over unbounded collections. This collection may grow unexpectedly large, enough so a call will not be able to iterate the entire collection in a single transaction because of gas limits. This results in Denial of Service in the form of contract soflocking, calling those functions will always fail unless the gas limit increases or the collection shrinks.
We recommend revisiting these loops, either documenting why you believe their growth is bounded, replacing them with bounded versions or removing them altogether replacing the functionality with something equivalent.

Minor severity

Remove non calling contracts from the whitelist

Some contracts do not call contracts that require whitelisting. This includes the tokens, MoCBProxManager and BtcPriceProvider. This last contract can be especially concerning as it will be developed independently from this project as an oracle, and may escalate to a bigger vulnerability if not checked correctly.
We recommend removing these contracts from the whitelist.

Usage of block numbers to approximate days

The contract MoCInrate uses block numbers to approximate a day duration by calculating the average expected block time. Block time is variable, so it will definitely change over time.
Timestamps are a better option for this, while they can be manipulated by the miners, they must be strictly increasing. This means a block can't have a lower timestamp than the previous block. Moreover a much higher manipulated timestamp will result in an orphaned block, since blocks with legitimate timestamps cannot be put after it, therefore the chain will not increase until the manipulated timestamp is reached.
We recommend using timestamps as they are secure as long as you use them to calculate greater periods of time, like days in this case.

Old solidity version

The solidity compiler version of 0.4.24 required in the pragmas is 11 months old at the time of writing this document. 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.

Missing message in requires

The *require()* statement has an optional message parameter that will return in case of failure of the testing condition. The message is already present in the majority of the contracts, but in some of them it is missing.

- MoCWhitelist.sol: It doesn't have a message in any of the requires. For example in function *add*

```
function add(address account) public onlyOwner {
  require(account != address(0));
  require(!isWhitelisted(account));
}
```

- MoCState.sol: The function *setMaxDiscountRate* do not have a message

```
function setMaxDiscountRate(uint256 rate) public onlyWhitelisted(msg.sender) {
  require(rate < mocLibConfig.ratePrecision);

  bproMaxDiscountRate = rate;
}
```

We recommend to always include the error message for consistent usage.

Enhancements

Document the rest of the functions

Most of the functions in the project are properly documented, but some aren't and others have outdated documentation. For example the function **alterRedeemRequestAmount** in MocSettlement specifies it returns an amount, but it doesn't return anything.
We recommend documenting the rest of the functions.

Optimization

Replace use of strings

In MoCBucketContainer a string is used as a key to a mapping. It should be possible to use a bytes32 instead to save some gas. Strings are not native to the EVM and use arrays of variable size internally and the contract needs a larger runtime.

Conclusion

We found the contracts to be simple and straightforward and have a good amount of documentation.

The use of whitelisting through a modifier simplifies the analysis of functions interactions between them because it prevents the attack of an unrelated third party.

Although the *transitionState* modifier is very complex by its nature, it simply acts as a passthrough after the state verification is not *Liquidated*. And in that case it forces the liquidation of the contract.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the Money on Chain project since Coinfabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.