SOMISH
Blockchain Labs

Smart Contract Audit Report
Akiva Capital

February 11th, 2020

SOMISH
Blockchain Labs

Index

Summary

Audit Report prepared by Somish Blockchain Labs for Akiva Capital smart contract.

Process and Delivery

Two (2) independent experts performed an unbiased and isolated audit of the code below. The debrief took place on February 1st, 2020, and the final results are presented here.

Audited Files

The following contract(s) were covered during the audit:

- **FraFactory.sol**.
- **Agreement.sol**.
- **Claimable.sol**.
- **Administrable.sol**.
- **McdWrapper.sol.**
- **FraQueries.sol.**
- **Initializable.sol.**
- **RaySupport.sol.**
- **Config.sol.**

Client Documents

The following document(s) were provided by the client for the purpose of the audit:

- Main - Document Outlining Important details of Akiva Capital dated 23rd June 2019.
- Supporting document outlining changes of Akiva Capital dated February 1st, 2020.

Notes

Audit was performed on commit dc6f48f71a56ed124b18571f7c60bb4306dd53f2 and pragma version 0.5.12

Intended Behavior

1. Product context

1.1 Product Perspective

Distributed Finance (DeFi) based product on Ethereum blockchain and MakerDAO, allowing:

- Leverage of crypto investment positions with loans with hedged interest rate risk
- Fixed-income investment

1.2 Functional definition

The current version implements the capability to open/close FRA contracts between lenders and loan seekers.

1.3 User Classes and Characteristics

- Fixed income seekers, aka lenders, aka DAI users
- Variable-rate loan borrowers, aka crypto position owners, aka loan takers, aka CDP users
- Admins (ACH)

2. Single contract lifecycle

2.1 Contract initiation

Akiva assumes that a CDP user (borrower) already has their position wrapped. They go to the FRA contract and input all the necessary parameters such as “minimum collateralization ratio” “discount amount” “expiry date” (max expiry date is 1 year from current date.) and ”fixed interest rate” and whatever else is necessary (if you think of some other variable or parameter that Akiva didn’t describe here can be assumed as inputted by the CDP user (“borrower”))

- Many of the input parameters will be inferred from the CDP wrapper contract, such as debt amount (denominated in DAI), collateral type, and collateral value, we assume this exists already

- Once the CDP user has collateralized the contract with their position they can call a function called open() this function enables a counterparty to come along and collateralize the contract. The contract cannot be split into many pieces, there is always 1 CDP user and 1 DAI user. Only the person who created the contract can call the open function. (This is as opposed to the check() function which can be called by anyone)

- There also exists a function called close() which can only be called by the creator of the contract. It can only be called if there has not been found a counterpart yet, this allows them to receive their collateral back and close the contract
- At this point, the contract waits to receive input from a DAI user, once a DAI user has selected the contract and sent in the debt amount in DAI the contract is considered to have begun. If the DAI user inputs too little DAI the contract does not begin, it should wait for them to send the appropriate amount of DAI as queried from the CDP wrapper contract.

2.2 Algorithm Overview

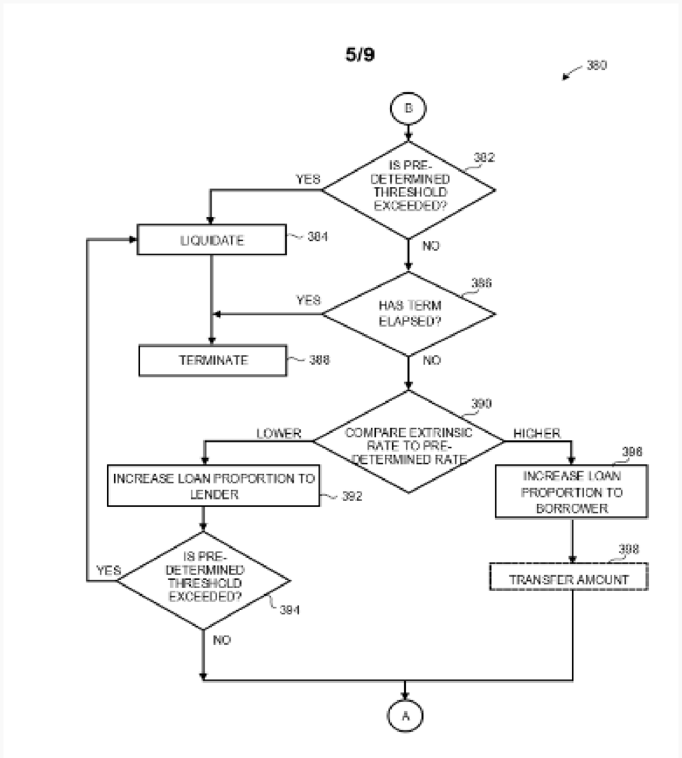

5/9
380
B
IS PRE-DETERMINED THRESHOLD EXCEEDED?
382
YES
NO
LIQUIDATE
384
HAS TERM ELAPSED?
386
YES
NO
TERMINATE
388
COMPARE EXTRINSIC RATE TO PRE-DETERMINED RATE
390
LOWER
HIGHER
INCREASE LOAN PROPORTION TO LENDER
392
INCREASE LOAN PROPORTION TO BORROWER
396
IS PRE-DETERMINED THRESHOLD EXCEEDED?
394
YES
NO
TRANSFER AMOUNT
398
A

FIG. 3C

- The "**minimum collateralization ratio**" determines at what point the CDP gets sold off. So if we assume it is set at 110%, then the value of the CDP must always be 1.1 times the amount that the borrower owes the lender.

- A “**CDP wrapper**” is a contract that holds the ownership of the CDP. Ownership can be transferred only by the owner of that contract. It also contains oracles to get the current value of the CDP and the current debt size and the collateral type.
- A “**keeper**” is someone who buys the CDP in case of liquidation (arbitrageurs).
- “**Predetermined threshold**” is the liquidation ratio. This is the minimum value of the CDP as compared to the amount borrower owes lender.
- **“Terminate”** is when contract reaches expiry and amounts must be settled.
- “**Liquidate**” is when it is sold off for “discount amount” and appropriate proceeds are sent off to both sides. A sufficient amount to cover the lenders interest and any leftover for the borrower.
- “**Compare rates**” this is the comparison of the contracts internal rate and the external DSR”.

2.3 Contract end or liquidation

Loan borrowers do not receive their collateral back until either the contract has expired, or the check() function has been called and it is found that the CDP user owes close to as much DAI as their CDP is worth in which case it is sold off to the first bidder at “discount amount” discounted from the true value of the collateral. Hence if the CDP is worth a thousand dollars and the discount amount is 0.03 (or 3 percent), then it is sold off for 9,700 DAI.
The selling off of the collateral works similarly as the DAI users collateralization of the contract. It goes to the first person to deposit the necessary amount. They have to figure out a way to restrict both of these opportunities to one user (lender/keeper (see bottom definition of keeper)). Akiva does not allow more than one user to bid on the collateral or to become a counterpart of a contract. It always goes to the first one to do it. Akiva does not know how to implement that but they imagine it should not be too hard.

2.4 Approval

Every time a CDP user begins to try to use a contract it requires approval from “owner”.
Owner is some user at Akiva Capital Holdings. The contracts are closed source.
It must be done in such a way that the CDP user (borrower) cannot replace the owner’s address in the contract code (thereby bypassing the approval). This should not be too difficult because we will always deploy from byte code and not source code. So the owner’s approval must be built into a function of the contract. (Since functions are not made available like data) We need to avoid the scenario where people are deploying contracts ACH approval.

2.5 Data stored

- The contract stores many types of Data, first of all, is the **addresses of both counterparties**. It also stores the debt amount, the collateral value, the historical DSR levels, the fixed-rate, and the amounts one of the sides owes to the other (if any). It also stores how long after the contract was instantiated until was executed. It also stores the minimum collateralization ratio. All of these values can be queried by anyone.

- In the case of liquidation, the contract stores how long it took to find a buyer for the CDP. If it takes more than a week it is sent to the DAI user (lender)

- It also stores whether the contract is “open” or “close”, or “collateralized” open and close are based on the functions talked about above, and collateralized means that there are two counterparts and the contract is live.

Issues Found

Critical

1. _updateAgreementState() function can be called by anyone.

This function **_updateAgreementState**, which takes a flag _isLastUpdate as a boolean parameter, can be called by anyone. If this function is called with ‘true’ as parameter, it shall terminate the agreement accordingly. Also this function can be called at any stage of agreement contract, there is no check for the current status of the agreement whether it is open, pending, active or closed. Unintentional calls to this function can result in an unwanted state of the contract.

Recommendation
Consider making the function internal if it only needs to be called by **FraFactory** contract or if this function needs to be open then perform input validations and add multiple checks for when and who can call this function.

2. User’s funds can get stuck due to improper handling of low level calls.

Due to unchecked low level calls in **McdWrapper.sol** critical situations can occur where user’s funds can get stuck.

For eg :- In **Config.sol** mapping named **collateralsEnabled** stores whether a type of collateral is enabled by the platform or not. Issue can arise if a collateral is enabled by the platform but disabled by MCD contracts for any reason. In this case, as the borrower's collateral is already possessed by the Agreement contract, transaction to Mcd contract will likely fail when called by lender in **matchAgreement()** function. Since the return value of low level calls is not handled properly the external call will fail silently and a garbage **cdpId** will be stored in the Agreement contract. In this case, the borrower's collateral will get stuck in **Agreement.sol**.
Recommendation
Ensure a check on low level external calls to ensure a valid **cdpId**.

Major

1. Incomplete functionality - liquidation.

As per the requirement document, the project should sell the liquidated CDP that falls below the threshold limit to another user at a discounted price. We couldn't find any concept of keeper or selling of CDP in the contract's implementation.

2. The cron job will stop working after a point of time due to breach of gas limits.

The function **autoRejectAgreements()** iterates from **0** to **agreementList.length,** and may result in breach of block gas limits after addition of several records in **agreementList,** resulting in failure of cron job execution.
Recommendation
Consider implementing loops that can be called in multiple transactions or assert that an array never grows larger than what fits in a single block.

3. Too much Admin/Owner dependency.

Multiple functions in **FraFactory.sol** like **rejectAgreement()**, **batchRejectAgreements()**, **blockAgreement()** and **removeAgreement()** are protected by modifier **onlyAdmin** which means any admin can call these functions at their will. Also in case of **Config.sol** owner (deployer) of contract can easily call setter functions of the contract and can manipulate sensitive state variables.
For eg :- Owner can change **injectionThreshold** and **riskyMargin** for the platform anytime, these values are used in **_updateAgreementState()** and **_monitorRisky()** functions respectively at the time of execution. If **injectionThreshold** is set too high, the borrower's and lender's internal balance may never get updated. If **riskyMargin** is set too low, the Agreement will be marked as Risky.

Recommendation
Consider implementing a governance model or a multisig wallet for Admin's operations. Also all config parameters should be stored inside the agreement contract at the time of agreement between borrower and lender.

4. updateAgreement() function should be callable by anyone.

Since the function **updateAgreement()** is critical for handling the agreement and has all checks in place with no input parameters, the function should be kept external, open to all. The requirement document also mentions that the **updateAgreement()** or **check()** function should be callable by anyone but it is protected by **onlyContractOwner** modifier. Hence, currently this function can only be called by FraFactory contract via Admin.

Recommendation
Consider removing the **onlyContractOwner** modifier as all necessary checks are already implemented inside the function.

Minor

1. ERC-20 transfers should be wrapped in require statements.

The ERC20 standard defines that the **transfer()** function should return a boolean and although most implementations throw on failure, this behavior is not required. Therefore it is recommended to wrap transfer calls in require. You can also replace it with **safeTransfer()** from **SafeERC20.sol** as it is already wrapped in require statement.

Recommendation
Consider wrapping all ERC20 calls inside require statements.

2. Unchecked low-level calls.

There are no checks to ensure that low-level calls are executed successfully. A transaction does not revert in case a low-level call reverts so it is best practice to add a check for low-level calls.
Contract: McdWrapper.sol.

Recommendation
Consider adding a check for low-level calls.

3. Anyone can call initAgreement() in agreement.sol.

The function initAgreement() is declared as 'public' and hence any user can add data in the implementation contract. Ideally, the function should only be called by **FraFactory.sol.**

Recommendation
Consider restricting access to **FraFactory.sol**.

4. CollateralAmount limit should be based on Collateral type.

As every cryptocurrency has different rates, the limit for min/max collateral should be based on collateral type.

Recommendation
Consider maintaining a limit for the collateral amount based on collateral type.

5. In enum ClosedType, Cancelled value is used for multiple cases.

For the enum **ClosedType**, 'Cancelled' value is set when Agreement is cancelled by borrower using **cancelAgreement()** function as well as when Agreement is rejected by admin using **rejectAgreement()** function.

Recommendation
Consider adding another enum value like 'Rejected' for **ClosedType**.

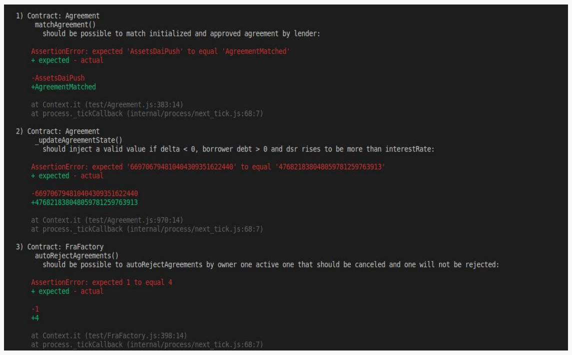
6. Pulling out of funds before expiration period is not implemented.

There is no mechanism present in which a user can pull out his/her funds before the expiration period. Generally in DeFi projects there exists a mechanism by which a user can pull out his/her funds in case of emergency or at his will. Currently the Agreement can only be terminated in case when expiration period is over or in the case of collateral liquidation.

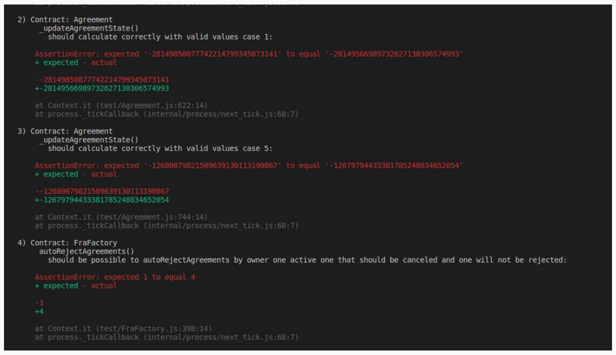
Recommendation
Allow withdrawal of funds before termination of agreement. The withdrawing party can be penalised by some amount or percentage in order to discourage withdrawals or early termination.

7. Failing test cases due to synchronisation issues.

Current implementation of test cases for the smart contracts have some synchronisation issues in the serial execution of test cases. Due to this, different test cases fail each time the test suite is executed.



```
1) Contract: Agreement
     matchAgreement()
       should be possible to match initialized and approved agreement by lender:

    AssertionError: expected 'AssetsDaiPush' to equal 'AgreementMatched'
    + expected - actual

    -AssetsDaiPush
    +AgreementMatched

    at Context.it (test/Agreement.js:383:14)
    at process._tickCallback (internal/process/next_tick.js:68:7)

2) Contract: Agreement
     _updateAgreementState()
       should inject a valid value if delta < 0, borrower debt > 0 and dsr rises to be more than interestRate:

    AssertionError: expected '669706794810404309351622440' to equal '476821838048059781259763913'
    + expected - actual

    -669706794810404309351622440
    +476821838048059781259763913

    at Context.it (test/Agreement.js:970:14)
    at process._tickCallback (internal/process/next_tick.js:68:7)

3) Contract: FraFactory
     autoRejectAgreements()
       should be possible to autoRejectAgreements by owner one active one that should be canceled and one will not be rejected:

    AssertionError: expected 1 to equal 4
    + expected - actual

    -1
    +4

    at Context.it (test/FraFactory.js:398:14)
    at process._tickCallback (internal/process/next_tick.js:68:7)
```



```
2) Contract: Agreement
     _updateAgreementState()
       should calculate correctly with valid values case 1:

    AssertionError: expected '-28149850877742214799345073141' to equal '-28149566989732627130306574993'
    + expected - actual

    --28149850877742214799345073141
    +-28149566989732627130306574993

    at Context.it (test/Agreement.js:622:14)
    at process._tickCallback (internal/process/next_tick.js:68:7)

3) Contract: Agreement
     _updateAgreementState()
       should calculate correctly with valid values case 5:

    AssertionError: expected '-12680079821509639130113190867' to equal '-12679794433381785248834652054'
    + expected - actual

    --12680079821509639130113190867
    +-12679794433381785248834652054

    at Context.it (test/Agreement.js:744:14)
    at process._tickCallback (internal/process/next_tick.js:68:7)

4) Contract: FraFactory
     autoRejectAgreements()
       should be possible to autoRejectAgreements by owner one active one that should be canceled and one will not be rejected:

    AssertionError: expected 1 to equal 4
    + expected - actual

    -1
    +4

    at Context.it (test/FraFactory.js:398:14)
    at process._tickCallback (internal/process/next_tick.js:68:7)
```

Notes

1. Gas Optimizations.

- **Consider using smaller bits uint variables.**
  In **Agreement.sol** all integer variables are declared as uint256/int256 which uses 256 bits of storage on the contract. If used properly smaller sized variables can be packed together in a single storage slot which results in code optimization.

  Recommendation
  Consider using smaller bits sized variable for storing **cdpId**, **interestRate**, etc.

- **No need to initialize enum type Status with ‘All’.**
  As soon as the Agreement contract is initialized the value of enum variable **status** is changed to ‘Pending’ so there is no need to initialize it with ‘All’ and then change the value to ‘Pending’.

  Recommendation
  Consider removing the ‘All’ status from the enum type. This will also reduce gas cost as writing to storage variables cost the most gas.

- **Consider declaring function as external.**

In **FraFactory.sol**, most of the functions are declared as public. Since these functions will always be called externally consider changing the declaration to external instead of public.
As external functions cost less gas than public function especially in the case where function arguments are arrays because in public functions, Solidity immediately copies array arguments to memory, while external functions can read directly from calldata. Memory allocation is expensive, whereas reading from calldata is cheap.

Recommendation
Consider changing the function declarations to external.

- **Use of extra variables.**
  Boolean variable **isRisky** is declared as a state variable in **Agreement.sol** whether we didn't see any need to store that boolean value as it is just used in **_monitorRisky()** function for emitting an event.

  Recommendation
  Consider removing the variable if possible.

2. No way to upgrade implementation for already created agreements.

As the code creating proxy contracts for each agreement, consider adding function to upgrade implementations for already created agreements. It will help in cases where any loopholes are discovered later.

Recommendation
Consider adding a function that upgrades implementation addresses of all created agreements.

3. The owner has the privilege to change configAddr and agreementImpl whenever he wants.

As the owner has the privilege to change **configAddr** and **agreementImpl** whenever he wants, It will make the system dependent on the **owner**. In any decentralized application, it is not a best practice to have any kind of high privilege access to any individual.

Recommendation
Consider adding a governance like feature for upgrading **configAddr** and **agreementImpl**.

4. **Avoid using assembly language code wherever possible.**

   The usage of assembly is considered as error-prone and should be avoided wherever possible.

Closing Summary

Upon audit of Akiva Capital's smart contract, it was observed that the contracts contain critical, major and minor issues, along with several areas of notes.

We recommend that critical, major and minor issues should be resolved. Resolving the areas of notes is up to Akiva Capital's discretion. The notes refer to improving the operations of the smart contract.

Disclaimer

Somish Blockchain Labs's audit is not a security warranty, investment advice, or an endorsement of the Akiva Capital's platform or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Somish from legal and financial liability.

Somish Solutions Limited