QuillAudits

Audit Report
January, 2021

Contents

Introduction

This Audit Report mainly focuses on the overall security of AskoLend Smart Contracts. With this report, we have tried to ensure the reliability and correctness of their smart contract by a complete and rigorous assessment of their system's architecture and the smart contract codebase.

Auditing Approach and Methodologies applied

The Quillhash team has performed rigorous testing of the project starting with analysing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third party smart contracts and libraries.

Our team then performed a formal line by line inspection of the Smart Contract to find any potential issue like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

In Automated Testing, We tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was tested in collaboration of our multiple team members and this included -

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analysing the complexity of the code in-depth and detailed, manual review of the code, line-by-line.
- Deploying the code on testnet using multiple clients to run live tests.
- Analysing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analysing the security of the on-chain data.

Audit Details

Project Name: AskoLend
Languages: Solidity (Smart contract)

Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Securify, Mythril, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the Smart Contract System is secure, resilient and working according to the specifications. The audit activities can be grouped in the following three categories:

Security
Identifying security related issues within each contract and the system of contract.

Sound Architecture
Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality
A full review of the contract source code. The primary areas of focus include:

- Accuracy
- Readability
- Sections of code with high complexity

Issue Categories

Every issue in this report was assigned a severity level from the following:

High severity issues

Issues on this level are critical to the smart contract's performance/ functionality and should be fixed before moving to a live environment.

Medium severity issues

Issues on this level could potentially bring problems and should eventually be fixed.

Low severity issues

Issues on this level are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	0	7
Closed	9	9	2	7

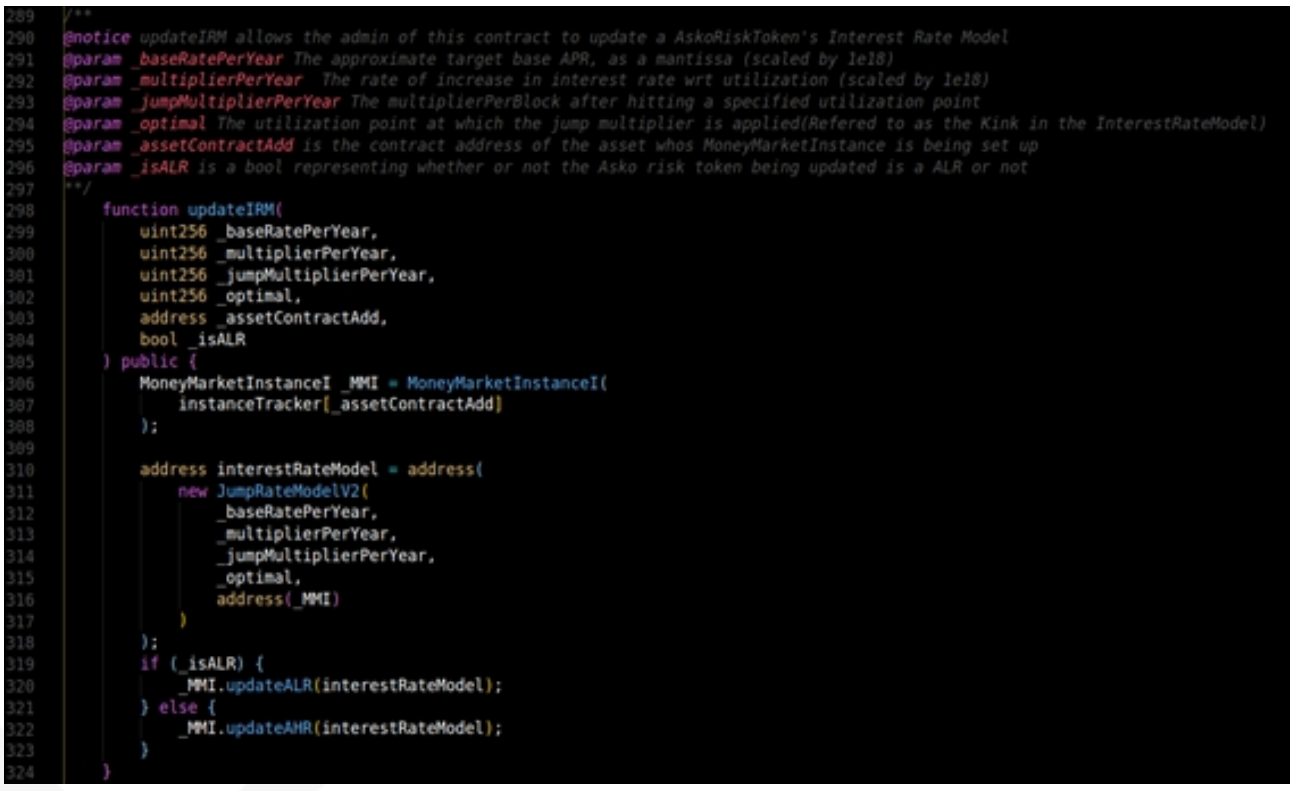
Manual Audit

For this section, the code was tested/read line by line by our developers. We also used Remix IDE's JavaScript VM and Kovan networks to test the contract functionality.

High level severity issues

1. **Money Market Control**

   The function updateIRM in this contract as per the comments can only be called by the admin of the contract. We did not find any role level permissions or require messages that restrict this public function, which means it can be called by anyone. If this is not a desired functionality, then it might be critical. We think adding the onlyMMI modifier should resolve this as used in the updateRR function which only the admin has access to as well.



```
/**
@notice updateIRM allows the admin of this contract to update a AskoRiskToken's Interest Rate Model
@param _baseRatePerYear The approximate target base APR, as a mantissa (scaled by 1e18)
@param _multiplierPerYear  The rate of increase in interest rate wrt utilization (scaled by 1e18)
@param _jumpMultiplierPerYear The multiplierPerBlock after hitting a specified utilization point
@param _optimal The utilization point at which the jump multiplier is applied(Refered to as the Kink in the InterestRateModel)
@param _assetContractAdd is the contract address of the asset whos MoneyMarketInstance is being set up
@param _isALR is a bool representing whether or not the Asko risk token being updated is a ALR or not
**/
    function updateIRM(
        uint256 _baseRatePerYear,
        uint256 _multiplierPerYear,
        uint256 _jumpMultiplierPerYear,
        uint256 _optimal,
        address _assetContractAdd,
        bool _isALR
    ) public {
        MoneyMarketInstanceI _MMI = MoneyMarketInstanceI(
            instanceTracker[_assetContractAdd]
        );

        address interestRateModel = address(
            new JumpRateModelV2(
                _baseRatePerYear,
                _multiplierPerYear,
                _jumpMultiplierPerYear,
                _optimal,
                address(_MMI)
            )
        );
        if (_isALR) {
            _MMI.updateALR(interestRateModel);
        } else {
            _MMI.updateAHR(interestRateModel);
        }
    }
```

Status: Closed

2. **Money Market Instance**

The contract fails to compile initially with the following error:

```
localhost/AskoLend-
contracts/contracts/MoneyMarket
Instance.sol:190:9: TypeError:
Member "safeTransferFrom" not
found or not visible after
argument-dependent lookup in
contract IERC20.
asset.safeTransferFrom(msg.send
er, address(AHR), _amount); ^--
-----------------^
```

This is for all statements where asset.safeTransferFrom is used. This is mainly because the contract is missing the following statement:

using SafeERC20 for IERC20;

Adding this in the beginning of the contract should resolve the compilation errors.

Status: Closed

Medium level severity issues

1. **Money Market Control**

1.1 In function **updateIRM**, we should add a **require** assertion to check if **instanceTracker[_assetContractAdd]** exists in the beginning of the function. This will ensure that the function reverts if **instanceTracker[_assetContractAdd]** returns a 0 address and saves unnecessary gas losses which might happen in the current implementation if an unavailable asset is provided. We are having the same issues in the following functions as well:

- **updateRR**
- **setupAHR**
- **setUpALR**

Status: Closed

1.2 In functions **trackCollateralUp** and **trackCollateralDown**, we should add a require assertion to check if **collateralTracker**[_borrower][_ALR] exists or not and revert accordingly. Not checking this in the current implementation could cause the function to run successfully even though wrong borrower and _ALR addresses are provided.

Status: Closed

1.3 In the function **liquidateTrigger**, we should add a require assertion to check if **collateralTracker**[_borrower][address(_ALR)] exists or not and revert accordingly. Not checking this in the current implementation could cause the function to run successfully or cause unnecessary gas losses (if reverts later) if wrong addresses are provided.

Status: Closed

1.4 Any external functions having calls to other contracts are susceptible to reentrancy attacks. It is always better to handle this weakness although it is not much critical in our case. Here is a good article explaining reentrancy attacks: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21.
To deal with this we suggest using the **Mutex** method. Open zeppelin provides a nice library to deal with reentrancy vulnerabilities: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/utils/ReentrancyGuard.sol.

The following functions are susceptible to reentrancy in this contract:

- **whitelistAsset**
- **setUpAHR**
- **setUpALR**

Status: Closed

1.5. The following functions should be declared external as they are not used anywhere else in the contract. This saves gas on function call and contract deployment.

- **whitelistAsset**
- **setUpAHR**
- **setUpALR**
- **updateRR**
- **checkCollateralizedALR**
- **liquidateTrigger**
- **updateIRM**

Status: Closed

2. Money Market Instance

2.1 Any external functions having calls to other contracts are susceptible to reentrancy attacks. It is always better to handle this weakness although it is not much critical in our case. Here is a good article explaining reentrancy attacks: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21. To deal with this we suggest using the **Mutex** method. Open zeppelin provides a nice library to deal with reentrancy vulnerabilities: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/utils/ReentrancyGuard.sol.

The following functions are susceptible to reentrancy in this contract:

- **borrow**
- **repay**
- **lendToAHRpool**
- **lendToALRpool**

Status: Closed

2.2 The following functions should be declared external as they are not used anywhere else in the contract. This saves gas on function call and contract deployment.

- **_setUpAHR**
- **_setUpALR**
- **getAssetAdd**
- **viewLockedCollateralizedALR**
- **lendToAHRpoo**
- **lendToALRpool**
- **borrow**
- **repay**
- **liquidateAccount**
- **checkIfALR**
- **updateALR**
- **updateAHR**
- **setRRAHR**
- **setRRALR**

Status: Closed

3. AskoRisk Token

3.1 Any external functions having calls to other contracts are susceptible to reentrancy attacks. It is always better to handle this weakness although it is not much critical in our case. Here is a good article explaining reentrancy attacks: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21.
To deal with this we suggest using the **Mutex** method. Open zeppelin provides a nice library to deal with reentrancy vulnerabilities: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/utils/ReentrancyGuard.sol.

The following functions are susceptible to reentrancy in this contract:

- **redeem**
- **_liquidate**
- **Borrow**

Status: Closed

3.3 The following functions should be declared external as they are not used anywhere else in the contract. This saves gas on function call and contract deployment.

- **transfer**
- **balanceOfUnderlyingPrior**
- **getSupplyAPY**
- **getBorrowAPY**
- **mint**
- **redeem**
- **burn**
- **getAssetAdd**
- **getUSDCWorthOfART**
- **viewUSDCWorthOfART**
- **viewConvertToART**
- **_liquidate**
- **_updateInterestModel**

Status: Closed

Low level severity issues

1. The pragma versions used within these contracts are not locked. Consider using version 0.6.12 for deploying the contracts. Solidity source files indicate the versions of the compiler they can be compiled with.

```
pragma solidity ^0.7.0; // bad: compiles with 0.7.0 and above
pragma solidity 0.7.0; // good : compiles w 0.7.0 only
```

Status: Closed

2. We recommend getting rid of Factory contracts: ARTFactory and MoneyMarketFactory to save deployment and transaction gas costs. This is because they serve no other purpose other than deploying new child contracts which can be easily done by using the child contract interfaces directly to deploy from other contracts. The factory contract would serve useful if we want to keep track of all the child contracts deployed by maintaining a state in that contract like in the case of UniswapOracleFactory. This is a classic case where we want to optimise costs over design patterns if possible.

Status: Closed

3. Money Market Control

We recommend adding external update functions which can only be called by owner/admin to update the variables **Oracle, _MMF and ARTF**. This lets only the admin/owner update this variable later in case of updates and most importantly will let us change it in case there was some mistake in initialising it.

Status: Closed

4. Money Market Instance

4.1 The divisor variable is a constant public variable that can never be updated. We recommend adding an updateDivisor function which lets only the admin/owner update this variable later and most importantly will let us change it in case there was some mistake in initialising it.

```
constructor(
    address _assetContractAdd,
    address _oracleFactory,
    address _owner,
    address _ARTF,
    string memory _assetName,
    string memory _assetSymbol
) public {
    divisor = 10000;      stan36, a month ag
    assetName = _assetName;
    assetSymbol = _assetSymbol;
    UOF = UniswapOracleFactoryI(_oracleFacto
    MMF = MoneyMarketFactoryI(_owner);
    asset = IERC20(_assetContractAdd);
    ARTF = ARTFactoryI(_ARTF);
}
```

Status: Closed

4.2 We don't understand the purpose of the **lockedCollatera**l state variable. The only place it is updated is in the end of repay function:

lockedCollateral[msg.sender] = 0;

Even if we don't explicitly set the value to 0, by default the value is set to 0 (lockedCollateral[_anyAddress] is 0 if not explicitly set). So the function **viewLockedCollateralizedALR(address _account)** will always return 0 irrespective of what _account address is passed.

Status: Closed

4.3 We recommend looking at calculating the half of any value. There are number of places where we are dividing a number by 2 to calculate the half of that number like:

vars.halfVal = vars.assetAmountValOwed.div(2);

Here in case **vars.assetAmountValOwed** is an odd number will result in loss of amount value as solidity does not handle float values (5/2 gives 2 in case of solidity). We need to ensure if this is okay in our case. If we need the upper ceiling of the half value (3 in above example), then we need to convert it to the next even number and then calculate the half of it.

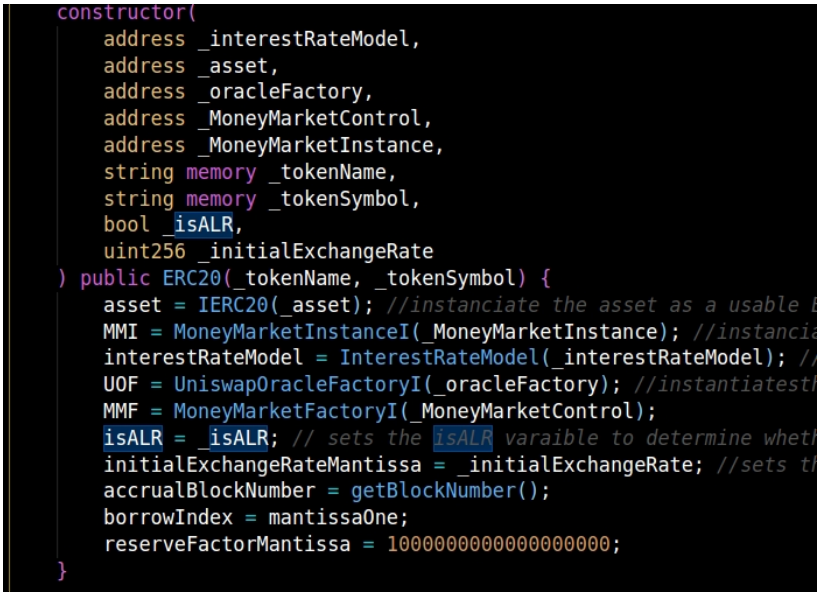
Status: Closed

4.4 We recommend adding update flows for variables **UOF, MMF, asset, ARTF** which cannot be changed once initialised. This will give us the flexibility to maybe update these later whenever there are changes in the corresponding contracts. These flows can be initiated from the Money market control when needed or by the contract owner.

Status: Closed

5. AskoRisk Token

5.1 We recommend adding update flows for all initialised variables wherever possible which cannot be changed once initialised. This will give us the flexibility to maybe update these later whenever there are changes in the corresponding contracts. These flows can be initiated from the Money market control when needed or by the contract owner.



```
constructor(
    address _interestRateModel,
    address _asset,
    address _oracleFactory,
    address _MoneyMarketControl,
    address _MoneyMarketInstance,
    string memory _tokenName,
    string memory _tokenSymbol,
    bool _isALR,
    uint256 _initialExchangeRate
) public ERC20(_tokenName, _tokenSymbol) {
    asset = IERC20(_asset); //instanciate the asset as a usable
    MMI = MoneyMarketInstanceI(_MoneyMarketInstance); //instanci
    interestRateModel = InterestRateModel(_interestRateModel); //
    UOF = UniswapOracleFactoryI(_oracleFactory); //instantiatest
    MMF = MoneyMarketFactoryI(_MoneyMarketControl);
    isALR = _isALR; // sets the isALR varaible to determine whet
    initialExchangeRateMantissa = _initialExchangeRate; //sets t
    accrualBlockNumber = getBlockNumber();
    borrowIndex = mantissaOne;
    reserveFactorMantissa = 1000000000000000000;
}
```

Status: Closed

5.2 We recommend making variables like **one, liquidationIncentiveMantissa** constants if they won't be updated later. This optimises gas costs on deployment and variable usage.

Status: Closed

Recommendations

1. Add events for most state changes. For example, after modifying balances for parties in _transfer function when adding rewards. Events should be fired with all state variable updates as good practice.

   Status: Closed

2. Follow solidity style guide for better readability: https://docs.soliditylang.org/en/v0.7.5/style-guide.html.
   For example, Functions should be grouped according to their visibility and ordered:
   constructor
   receive function (if exists)
   fallback function (if exists)
   external
   public
   internal
   private
   Within a grouping, place the view and pure functions last.

   Note: Linting violations can be easily fixed using linters like solhint.

   Status: Closed

3. All the “require” statements used in the contract should also specify error messages for easy debugging.

   Status: Closed

4. We recommend explicitly setting state visibility of all state variables wherever missing. Not providing any visibility defaults the variable to internal. This can help avoid ambiguity in contract code.

   Status: Closed

5. We recommend avoiding comparing variables to a boolean constant. This will save gas costs.
   Examples:

```
@notice onlyMMFactory is a modifier used to
**/
  modifier onlyMMI() {
      require(isMMI[msg.sender] == true);
      _;
  }
```

   This can be replaced with:
   require(isMMI[msg.sender]);

6. **Status: Closed**

   We recommend following the solidity naming conventions. There are a lot of naming in the contract that deviate from the conventions. For example, _enableStake_Bonus does not follow mixed case format.

7. **Status: Closed**

   We recommend removing public getter functions for fetching state variables and go with public state variables instead. This greatly will reduce deployment costs.
   For example you don't need the viewLockedCollateralizedALR function in the Money market instance contract if **lockedCollateral** mapping is made public.

   Status: Closed

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the AskoLend contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Summary

Altogether, the code is written, well documented and demonstrates effective use of abstraction, separation of concerns, and modularity. All high, medium and low severity issues as well as recommendations have been fixed by the AskoLend team and therefore the contract is good to be deployed on public networks as per the audit team's analysis.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com