Security Audit Report for Lista Lending

Date: April 3, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Lista
Target	Lista Lending

Version History

Version	Date	Description
1.0	April 3, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-12-04 04:11:06+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository of Lista Lending[1] of Lista. This audit focuses on the smart contracts located in the `src/folder` of the repository, excluding the following directories:

- src/moolah/mocks/*
- src/moolah-vault/mocks/*
- src/vault-allocator/mocks/*

Other contracts and source code files outside the `src/folder`, or within the excluded directories, are out of scope for this audit. The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process.

Project	Version	Commit Hash
Lista Lending	Version 1	423edc1dbb371de1f398d497815ec8757d49349b
	Version 2	96b0b210764b3b34b2e3d16cc8555f04646df424

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/lista-dao/moolah

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **five** potential security issues. Besides, we have **three** recommendations and **four** notes.

- Medium Risk: 1
- Low Risk: 4
- Recommendation: 3
- Note: 4

ID	Severity	Description	Category	Status
1	Medium	Potential inflation attacks	DeFi Security	Fixed
2	Low	Lack of validation checks in the `createMarket()` function	DeFi Security	Fixed
3	Low	Bypass of the bad debt handling mechanism in the `liquidate()` function	DeFi Security	Fixed
4	Low	Potential replay attacks due to the chain hard fork	DeFi Security	Confirmed
5	Low	Potential DoS risk in the `reallocate()` function	DeFi Security	Confirmed
6	-	Remove the improperly used and unused code	Recommendation	Confirmed
7	-	Revise the method used for the transfer of native tokens	Recommendation	Confirmed
8	-	Unify the use of the `_updateLastTotalAssets()` function for updating the variable `lastTotalAssets`	Recommendation	Confirmed
9	-	Potential centralization risks	Note	-
10	-	Return value of the functions `maxDeposit()/maxMint()`	Note	-
11	-	Potential griefing risk	Note	-
12	-	MoolahVault's assets may be redistributed through flash loans	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential inflation attacks

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `Moolah`, the use of `VIRTUAL_SHARES` (i.e., 1e6) and the rounding strategy in the asset/share conversion introduce potential vulnerabilities to inflation attacks in specific scenarios.

1. A malicious actor could perform a borrow operation of `1e6 - 1` shares in an empty market, resulting in zero borrow assets. This occurs because the calculation of borrow assets (i.e., `assets = shares.toAssetsDown()`) rounds down the result to zero (i.e., $1 \times (1e6-1)/1e6 = 0$). Consequently, the malicious actor could inflate the total borrow shares, blocking the `borrow()` function for other users.

```
function borrow(
  MarketParams memory marketParams,
  uint256 assets,
  uint256 shares,
  address onBehalf,
  address receiver
) external whenNotPaused nonReentrant returns (uint256, uint256) {
  Id id = marketParams.id();
  require(market[id].lastUpdate != 0, ErrorsLib.MARKET_NOT_CREATED);
  require(UtilsLib.exactlyOneZero(assets, shares), ErrorsLib.INCONSISTENT_INPUT);
  require(receiver != address(0), ErrorsLib.ZERO_ADDRESS);
  // No need to verify that onBehalf != address(0) thanks to the following authorization check.
  require(_isSenderAuthorized(onBehalf), ErrorsLib.UNAUTHORIZED);

  _accrueInterest(marketParams, id);

  if (assets > 0) shares = assets.toSharesUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);
  else assets = shares.toAssetsDown(market[id].totalBorrowAssets, market[id].totalBorrowShares);

  position[id][onBehalf].borrowShares += shares.toUint128();
  market[id].totalBorrowShares += shares.toUint128();
  market[id].totalBorrowAssets += assets.toUint128();
}
```

Listing 2.1: src/moolah/Moolah.sol

```
function toAssetsDown(uint256 shares, uint256 totalAssets, uint256 totalShares) internal pure returns (uint256) {
  return shares.mulDivDown(totalAssets + VIRTUAL_ASSETS, totalShares + VIRTUAL_SHARES);
}
```

Listing 2.2: src/moolah/libraries/SharesMathLib.sol

2. Similarly, a malicious actor could sequentially supply `1` asset and withdraw `1e6 - 1` shares to inflate the total supply assets due to the rounding down strategy used to perform share-to-asset conversion in the `withdraw()` function. As a result, the asset price is increased and the malicious actor could front-run any user's supply (via specifying the share amount), causing the user to pay more assets.

```
function withdraw(
  MarketParams memory marketParams,
  uint256 assets,
  uint256 shares,
  address onBehalf,
```

```
    address receiver
  ) external whenNotPaused nonReentrant returns (uint256, uint256) {
    Id id = marketParams.id();
    require(market[id].lastUpdate != 0, ErrorsLib.MARKET_NOT_CREATED);
    require(UtilsLib.exactlyOneZero(assets, shares), ErrorsLib.INCONSISTENT_INPUT);
    require(receiver != address(0), ErrorsLib.ZERO_ADDRESS);
    // No need to verify that onBehalf != address(0) thanks to the following authorization
        check.
    require(_isSenderAuthorized(onBehalf), ErrorsLib.UNAUTHORIZED);

    _accrueInterest(marketParams, id);

    if (assets > 0) shares = assets.toSharesUp(market[id].totalSupplyAssets, market[id].
        totalSupplyShares);
    else assets = shares.toAssetsDown(market[id].totalSupplyAssets, market[id].
        totalSupplyShares);

    position[id][onBehalf].supplyShares -= shares;
    market[id].totalSupplyShares -= shares.toUint128();
    market[id].totalSupplyAssets -= assets.toUint128();
```

Listing 2.3: src/moolah/Moolah.sol

```
function toAssetsDown(uint256 shares, uint256 totalAssets, uint256 totalShares) internal
    pure returns (uint256) {
  return shares.mulDivDown(totalAssets + VIRTUAL_ASSETS, totalShares + VIRTUAL_SHARES);
}
```

Listing 2.4: src/moolah/libraries/SharesMathLib.sol

Impact The improper use of `VIRTUAL_SHARES` and rounding strategy could lead to inflation attacks affecting users' operations.

Suggestion Revise the code logic accordingly.

2.1.2 Lack of validation checks in the `createMarket()` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Moolah` contract, the `createMarket()` function allows users to create market with the customized market configuration such as `loanToken`, `collateralToken`, and `oracle`. However, without sufficient validation checks, created markets could become invalid or even malicious. Specifically, the `createMarket()` function does not check whether the `loanToken` and `collateralToken` are valid ERC20 tokens (i.e., non-zero address).

```
function createMarket(MarketParams memory marketParams) external {
  Id id = marketParams.id();
  require(isIrmEnabled[marketParams.irm], ErrorsLib.IRM_NOT_ENABLED);
  require(isLltvEnabled[marketParams.lltv], ErrorsLib.LLTV_NOT_ENABLED);
  require(market[id].lastUpdate == 0, ErrorsLib.MARKET_ALREADY_CREATED);
```

```
  require(marketParams.oracle != address(0), ErrorsLib.ZERO_ADDRESS);

  // Safe "unchecked" cast.
  market[id].lastUpdate = uint128(block.timestamp);
  idToMarketParams[id] = marketParams;

  emit EventsLib.CreateMarket(id, marketParams);

  // Call to initialize the IRM in case it is stateful.
  if (marketParams.irm != address(0)) IIrm(marketParams.irm).borrowRate(marketParams, market[id
      ]);
}
```

Listing 2.5: src/moolah/Moolah.sol

Additionally, the configured oracle is used to determine the collateral price by querying `basePrice` and `quotaPrice` in the `getPrice()` function. However, there are no validation checks on the configured oracle in the function `createMarket()`, which could result in the use of an incorrect or malicious oracle. Specifically, considering a previous incident, if the decimals of the base and quota assets in the customized oracle are not aligned, it could lead to a potential loss of funds.

```
function getPrice(MarketParams memory marketParams) public view returns (uint256) {
  IOracle _oracle = IOracle(marketParams.oracle);
  uint256 baseTokenDecimals = IERC20Metadata(marketParams.collateralToken).decimals();
  uint256 quotaTokenDecimals = IERC20Metadata(marketParams.loanToken).decimals();
  uint256 basePrice = _oracle.peek(marketParams.collateralToken);
  uint256 quotaPrice = _oracle.peek(marketParams.loanToken);

  uint256 scaleFactor = 10 ** (36 + quotaTokenDecimals - baseTokenDecimals);
  return scaleFactor.mulDivDown(basePrice, quotaPrice);
}
```

Listing 2.6: src/moolah/Moolah.sol

Impact The lack of validation checks in the function `createMarket()` could lead to the creation of invalid markets and the potential loss of funds due to the use of invalid (or malicious) oracle.

Suggestion Add validation checks for significant parameters in the function `createMarket()`.

Note The project team left a note to emphasize that the decimals of the base and quota assets in the customized oracle should be aligned to 1e8.

2.1.3 Bypass of the bad debt handling mechanism in the `liquidate()` function

Severity Low

Status Fixed

Introduced by `Version 1`

Description In the contract `Moolah`, the function `liquidate()` allows whitelisted liquidators to liquidate a borrower's position by specifying the exact amount of `seizedAssets` or `repaidShares`. A bad debt arises when the collateral assets in a position cannot fully cover the loan assets.

However, the function `liquidate()` only handles bad debt when the position's collateral is fully seized (i.e., `position[id][borrower].collateral == 0`). This design introduces a potential risk of leaving the market in an unhealthy state by allowing a small amount of collateral (e.g., 1 wei of a collateral token) to remain in the position, thereby bypassing the bad debt handling mechanism.

```
function liquidate(
    MarketParams memory marketParams,
    address borrower,
    uint256 seizedAssets,
    uint256 repaidShares,
    bytes calldata data
  ) external whenNotPaused nonReentrant returns (uint256, uint256) {

    // ...

    if (position[id][borrower].collateral == 0) {
      badDebtShares = position[id][borrower].borrowShares;
      badDebtAssets = UtilsLib.min(
        market[id].totalBorrowAssets,
        badDebtShares.toAssetsUp(market[id].totalBorrowAssets, market[id].totalBorrowShares)
      );

      market[id].totalBorrowAssets -= badDebtAssets.toUint128();
      market[id].totalSupplyAssets -= badDebtAssets.toUint128();
      market[id].totalBorrowShares -= badDebtShares.toUint128();
      position[id][borrower].borrowShares = 0;
    }

    // ...
}
```

Listing 2.7: src/moolah/Moolah.sol

Impact The market could become unhealthy if bad debts are intentionally left unhandled.

Suggestion Add a health check at the end of the function `liquidate()` to ensure that bad debts are handled even when the position's collateral is not fully seized.

Note According to the design, the liquidator is not concerned about bad debt. Team promises the fully liquidation if the market is listed in their website.

2.1.4 Potential replay attacks due to the chain hard fork

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `constructor()` of the `Moolah` contract, the `EIP712_DOMAIN_TYPEHASH` is computed using the value of `block.chainid` and `address(this)` to verify users' signatures in the function `setAuthorizationWithSig()`. However, the fixed value of `EIP712_DOMAIN_TYPEHASH` could potentially result in signature replay attacks when a chain hard fork occurs (i.e., the value of `block.chainid` changes). As a result, any signature generated with the previous `block.chainid` could be replayed on the hard-forked chain.

```
constructor() {
    _disableInitializers();
    DOMAIN_SEPARATOR = keccak256(abi.encode(DOMAIN_TYPEHASH, block.chainid, address(this)));
  }
```

Listing 2.8: src/moolah/Moolah.sol

```
function setAuthorizationWithSig(Authorization memory authorization, Signature calldata signature) external {
  /// Do not check whether authorization is already set because the nonce increment is a desired side effect.
  require(block.timestamp <= authorization.deadline, ErrorsLib.SIGNATURE_EXPIRED);
  require(authorization.nonce == nonce[authorization.authorizer]++, ErrorsLib.INVALID_NONCE);

  bytes32 hashStruct = keccak256(abi.encode(AUTHORIZATION_TYPEHASH, authorization));
  bytes32 digest = keccak256(bytes.concat("\x19\x01", DOMAIN_SEPARATOR, hashStruct));
  address signatory = ecrecover(digest, signature.v, signature.r, signature.s);

  require(signatory != address(0) && authorization.authorizer == signatory, ErrorsLib.INVALID_SIGNATURE);

  emit EventsLib.IncrementNonce(msg.sender, authorization.authorizer, authorization.nonce);

  isAuthorized[authorization.authorizer][authorization.authorized] = authorization.isAuthorized;
```

Listing 2.9: src/moolah/Moolah.sol

Impact Signatures generated before the hard fork could be replayed on the new chain due to the lack of validation on the current `block.chainid` during signature verification.

Suggestion Add validation checks in the function `setAuthorizationWithSig()` to prevent replay attacks.

2.1.5 Potential DoS risk in the `reallocate()` function

Severity Low

Status Confirmed

Introduced by Version 1

Description In the contract `MoolahVault`, the `ALLOCATOR` role is allowed to invoke the function `reallocate()` to manage users' collateral across enabled markets. The function `reallocate()` requires that all withdrawn assets be fully supplied to other valid markets. However, the invocation of the function `reallocate()` is potentially vulnerable to a front-running attack, leading to a DoS issue. Specifically, when the `ALLOCATOR` role invokes the function `reallocate()` with improper input allocations (e.g., the last allocation is not set to max supply), malicious actors could front-run this invocation via a deposit or withdrawal, causing the reallocation to fail.

```
function reallocate(MarketAllocation[] calldata allocations) external onlyRole(ALLOCATOR) {
    uint256 totalSupplied;
```

```
    uint256 totalWithdrawn;
    for (uint256 i; i < allocations.length; ++i) {
      MarketAllocation memory allocation = allocations[i];
      Id id = allocation.marketParams.id();

      (uint256 supplyAssets, uint256 supplyShares, ) = _accruedSupplyBalance(allocation.
          marketParams, id);
      uint256 withdrawn = supplyAssets.zeroFloorSub(allocation.assets);

      if (withdrawn > 0) {
        if (!config[id].enabled) revert ErrorsLib.MarketNotEnabled(id);

        // Guarantees that unknown frontrunning donations can be withdrawn, in order to disable
            a market.
        uint256 shares;
        if (allocation.assets == 0) {
          shares = supplyShares;
          withdrawn = 0;
        }

        (uint256 withdrawnAssets, uint256 withdrawnShares) = MOOLAH.withdraw(
          allocation.marketParams,
          withdrawn,
          shares,
          address(this),
          address(this)
        );

        emit EventsLib.ReallocateWithdraw(_msgSender(), id, withdrawnAssets, withdrawnShares);

        totalWithdrawn += withdrawnAssets;
      } else {
        uint256 suppliedAssets = allocation.assets == type(uint256).max
          ? totalWithdrawn.zeroFloorSub(totalSupplied)
          : allocation.assets.zeroFloorSub(supplyAssets);

        if (suppliedAssets == 0) continue;

        uint256 supplyCap = config[id].cap;
        if (supplyCap == 0) revert ErrorsLib.UnauthorizedMarket(id);

        if (supplyAssets + suppliedAssets > supplyCap) revert ErrorsLib.SupplyCapExceeded(id);

        // The market's loan asset is guaranteed to be the vault's asset because it has a non-
           zero supply cap.
        (, uint256 suppliedShares) = MOOLAH.supply(allocation.marketParams, suppliedAssets, 0,
           address(this), hex"");

        emit EventsLib.ReallocateSupply(_msgSender(), id, suppliedAssets, suppliedShares);

        totalSupplied += suppliedAssets;
      }
    }
```

```
    if (totalWithdrawn != totalSupplied) revert ErrorsLib.InconsistentReallocation();
}
```

Listing 2.10: src/moolah-vault/MoolahVault.sol

Impact The invocation of the `reallocate()` function may fail due to a front-running attack.

Suggestion Revise the code logic accordingly.

2.2 Additional Recommendation

2.2.1 Remove the improperly used and unused code

Status Confirmed

Introduced by `Version 1`

Description The `GUARDIAN` role is set but not used in the contract `MoolahVault`. It is recommended to remove the unused role for better readability and gas optimization.

```
bytes32 public constant GUARDIAN = keccak256("GUARDIAN"); // manager role
```

Listing 2.11: src/moolah-vault/MoolahVault.sol

```
_setRoleAdmin(GUARDIAN, MANAGER);
```

Listing 2.12: src/moolah-vault/MoolahVault.sol

Moreover, there are several improperly used/unused variables, events, errors, functions, interfaces, and contracts in the protocol. It is recommended to revise or remove them for better code readability.

- **src/moolah/libraries/EventsLib.sol**
  - Unused events:
    - SetOwner()
- **src/moolah-vault/interfaces/IMoolahVault.sol**
  - Unused interfaces:
    - IOwnable
- **src/moolah-vault/libraries/ConstantsLib.sol**
  - Unused constants:
    - MAX_TIMELOCK, MIN_TIMELOCK
- **src/moolah-vault/libraries/ErrorsLib.sol**
  - Unused errors:
    - NotCuratorNorGuardianRole, AboveMaxTimelock, BlowMinTimelock, TimelockNotElapsed
  - Improperly used errors:
    - AlreadyPending, PendingRemoval, InvalidMarketRemovalTimelockNotElapsed
- **src/moolah-vault/libraries/EventsLib.sol**
  - Unused events:

  - SubmitTimelock, SetTimelock, SubmitGuardian, SetGuardian, SubmitCap, SubmitMarketRemoval, SetCurator, SetIsAllocator, RevokePendingTimelock
  - RevokePendingCap, RevokePendingGuardian, RevokePendingMarketRemoval
  - CreateMoolahVault
- **src/moolah-vault/libraries/PendingLib.sol**
  - This contract is unused

Suggestion Revise or remove the mentioned code.

2.2.2 Revise the method used for the transfer of native tokens

Status Confirmed

Introduced by `Version 1`

Description In the function `transferFee()` of the contract `VaultAllocator`, the `.transfer()` is used to transfer native tokens. It is recommended to use `.call()` due to the gas limitation (i.e., 2300) of the method `.transfer()`.

```
function transferFee(address vault, address payable feeRecipient) external
     onlyAdminOrVaultOwner(vault) {
    uint256 claimed = accruedFee[vault];
    accruedFee[vault] = 0;
    feeRecipient.transfer(claimed);
    emit EventsLib.TransferFee(msg.sender, vault, claimed, feeRecipient);
}
```

Listing 2.13: src/vault-allocator/VaultAllocator.sol

Suggestion Use the `.call()` method for native token transfers.

2.2.3 Unify the use of the `_updateLastTotalAssets()` function for updating the variable `lastTotalAssets`

Status Confirmed

Introduced by `Version 1`

Description The function `_updateLastTotalAssets()` is designated to update the value of `lastTotalAssets`. It is recommended to unify the update of the variable `lastTotalAssets` using the function for better code readability.

```
function deposit(uint256 assets, address receiver) public override returns (uint256 shares) {
    uint256 newTotalAssets = _accrueFee();

    // Update ‘lastTotalAssets‘ to avoid an inconsistent state in a re-entrant context.
    // It is updated again in ‘_deposit‘.
    lastTotalAssets = newTotalAssets;

    shares = _convertToSharesWithTotals(assets, totalSupply(), newTotalAssets, Math.Rounding.
        Floor);

    _deposit(_msgSender(), receiver, assets, shares);
}
```

Listing 2.14: src/moolah-vault/MoolahVault.sol

```
function mint(uint256 shares, address receiver) public override returns (uint256 assets) {
    uint256 newTotalAssets = _accrueFee();

    // Update `lastTotalAssets` to avoid an inconsistent state in a re-entrant context.
    // It is updated again in `_deposit`.
    lastTotalAssets = newTotalAssets;

    assets = _convertToAssetsWithTotals(shares, totalSupply(), newTotalAssets, Math.Rounding.Ceil);

    _deposit(_msgSender(), receiver, assets, shares);
}
```

Listing 2.15: src/moolah-vault/MoolahVault.sol

Suggestion Use the `_updateLastTotalAssets()` function to update `lastTotalAssets`.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description Several protocol roles could conduct privileged operations, which introduces potential centralization risks. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol. For example, the admin role of each contract has the authority to upgrade the contract and can upgrade the contract to a malicious contract.

It is worth noting that the `BOT` role in `Liquidator` contract must be trusted. Otherwise, the bot can conduct a sandwich attack by setting the slippage protection to be zero and steal the funds in the `Liquidator` contract accordingly. Furthermore, a malicious bot may deliberately delay liquidating debts to cause bad debts as the liquidation operation of contract `Moolah` is now restricted to the `BOT`.

Currently, the `InterestRateModel` is restricted to a whitelist. However, if in the future the interest rate model is not restricted to a whitelist, it will be possible for a malicious user to conduct re-enter attacks in the contract `MoolahVault`.

Feedback from the project We will try our best to ensure that the privileged roles (e.g., `BOT`) are trustworthy.

2.3.2 Return value of the functions `maxDeposit()/maxMint()`

Introduced by `Version 1`

Description The functions `maxDeposit()` and `maxMint()` may yield a value exceeding the actual maximum deposit due to the potential duplicate markets in the `supplyQueue`. The project team should notify users to prevent the direct use of the value returned by the functions `maxDeposit()`/`maxMint()`.

2.3.3 Potential griefing risk

Introduced by `Version 1`

Description Due to the design that the interest is not charged in the same block, a malicious user can front-run other users' withdrawal operations by borrowing all the remaining liquidity with enough collateral. After that, the attacker can then back-run in the same block to repay without any interest accrued. This can cause the system to be in a state where the user's withdrawal cannot be completed. The same risk also exists in the operations of borrowing.

2.3.4 MoolahVault's assets may be redistributed through flash loans

Introduced by `Version 1`

Description When the withdraw and supply queues are in the same order in the contract `MoolahVault`, a user can flash loan to deposit first to reach the front market's cap and distribute the assets to the behind markets in the supply queue. After that, the user can withdraw in the same block to drain the assets deposited in the front market. In that way, users can redistribute the assets in the `MoolahVault` to make potential beneficial for himself.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS