BLOCKSEC

Security Audit
Report for Lista Lending Provider

Date: May 12, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Lista
Target	Lista Lending Provider

Version History

Version	Date	Description
1.0	May 12, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-05-15 11:58:53+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 [1] of Lista Lending Provider of Lista. Lista Lending adds a SlisBNB and BNB provider on top of Moolah and MoolahVault. SlisBNBProvider allows users to mint clisBNB while they deposit slisBNB collateral. BNBProvider allows users to participate in Moolah and MoolahVault using native BNB. Note this audit only focuses on the smart contracts in the following directories/files:

- src/moolah-vault/MoolahVault.sol
- src/moolah/Moolah.sol
- src/moolah/SlisBNBProvider.sol
- src/provider/BNBProvider.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Lista Lending Provider	`Version 1`	`79f5dd0694bdba6732abe39752865579035e71e8` `5345086f1d200483fcce4a4a9dbca2eacf47795a`
	`Version 2`	`b9b40d7b11126dde05ff93443759e469a902ecff`

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 can-

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

not 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.

1.3 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.

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

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

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

- High Risk: 3
- Medium Risk: 5
- Low Risk: 2
- Recommendation: 5
- Note: 3

ID	Severity	Description	Category	Status
1	High	Potentially immutable `userLpRate` due to the unset variable `exchangeRate`	DeFi Security	Fixed
2	High	Incorrect position synchronization in the `withdrawCollateral()` function	DeFi Security	Fixed
3	High	Loss of funds due to the incorrect use of the input `assets`	DeFi Security	Fixed
4	Medium	Lack of checks on the variable `_userLpRate` in the function `initialize()`	DeFi Security	Fixed
5	Medium	The `delegatee` can be maliciously reset	DeFi Security	Fixed
6	Medium	Incorrect calculation of the `wrapAmount` variable	DeFi Security	Fixed
7	Medium	Users can obtain LP tokens without supplying collateral	DeFi Security	Fixed
8	Medium	Failure of supplying `WBNB` as collateral	DeFi Security	Fixed
9	Low	Lack of checks on the variable `wrapAmount` in the `repay()` function	DeFi Security	Fixed
10	Low	Potential DoS due to the lack of callback implementations	DeFi Security	Confirmed
11	-	Lack of invoking function `_disableInitializers()`	Recommendation	Fixed
12	-	Lack of zero address checks	Recommendation	Fixed
13	-	Add state change checks in the `setUserLpRare()` function	Recommendation	Fixed
14	-	Unify the the use of the variable `RATE_DENOMINATOR`	Recommendation	Fixed
15	-	Add rescue token functionality in `SlisBNBProvider` and `BNBProvider` contracts	Recommendation	Confirmed
16	-	The minting and burning mechanism of LP tokens	Note	-
17	-	Potential centralization risks	Note	-

18	-	Sufficient capacity in the MPC wallets	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potentially immutable `userLpRate` due to the unset variable `exchangeRate`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `setUserLpRate()`, the variable `userLpRate` will only be updated if the variable `_userLpRate` satisfies the condition (i.e., `_userLpRate <= 1e18 && _userLpRate <= exchangeRate)`. However, since the variable `exchangeRate` is not set (i.e., defaulting to 0) in the contract `SlisBNBProvider`, the function `setUserLpRate()` can not be invoked when the variable `userLpRate` is greater than zero. As a result, the variable `userLpRate` is immutable.

```
316 function setUserLpRate(uint128 _userLpRate) external onlyRole(MANAGER) {
317   require(_userLpRate <= 1e18 && _userLpRate <= exchangeRate, "userLpRate invalid");
318
319   userLpRate = _userLpRate;
320   emit UserLpRateChanged(userLpRate);
321 }
```

Listing 2.1: src/moolah/SlisBNBProvider.sol

Impact The variable `userLpRate` is immutable.

Suggestion Add the logic to set the `exchangeRate`.

2.1.2 Incorrect position synchronization in the `withdrawCollateral()` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `SlisBNBProvider` contract, the `withdrawCollateral()` function allows users to withdraw collateral (i.e., `slisBNB`) on behalf of the address `onBehalf` and update (i.e., mint or burn) the LP tokens (i.e., `clisBNB`) based on mutated positions. However, the `_syncPosition()` function incorrectly updates the LP tokens for `msg.sender`, which can be different from the address `onBehalf`. As a result, the LP tokens of the address `onBehalf` may not be updated.

```
148 function withdrawCollateral(
149   MarketParams memory marketParams,
150   uint256 assets,
151   address onBehalf,
152   address receiver
153 ) external {
```

```
154   require(assets > 0, "zero withdrawal amount");
155   require(_isSenderAuthorized(onBehalf), "unauthorized sender");
156   require(marketParams.collateralToken == token, "invalid collateral token");
157
158   // withdraw from distributor
159   MOOLAH.withdrawCollateral(marketParams, assets, onBehalf, address(this));
160   // rebalance user's lpToken
161   _syncPosition(marketParams.id(), msg.sender);
162
163   // transfer token to user
164   IERC20(token).safeTransfer(receiver, assets);
165   emit Withdrawal(msg.sender, assets);
166 }
```

Listing 2.2: src/moolah/SlisBNBProvider.sol

Impact The LP tokens of the address `onBehalf` may not be updated.

Suggestion Use the address `onBehalf` instead of `msg.sender`.

2.1.3 Loss of funds due to the incorrect use of the input `assets`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `BNBProvider` contract, the `borrow()` function allows users to borrow `WBNB` from the `Moolah` contract by specifying either the expected amount of `assets` or `shares`. After borrowing, the `BNBProvider` contract unwraps the `WBNB` and transfers `BNB` to the user via the low-level call (line 160). However, the call incorrectly uses the input `assets` instead of `_assets`, leading to a loss of funds. Specifically, if a user borrows based on the input `shares` (i.e., with the input `assets` set to zero), they will receive no `BNB`.

```
142 function borrow(
143   MarketParams calldata marketParams,
144   uint256 assets,
145   uint256 shares,
146   address onBehalf,
147   address payable receiver
148 ) external returns (uint256 _assets, uint256 _shares) {
149   // No need to verify assets and shares, as they are already verified in the Moolah contract.
150   require(marketParams.loanToken == address(WBNB), "invalid loan token");
151   require(isSenderAuthorized(msg.sender, onBehalf), ErrorsLib.UNAUTHORIZED);
152
153   // 1. borrow WBNB from moolah
154   (_assets, _shares) = MOOLAH.borrow(marketParams, assets, shares, onBehalf, address(this));
155
156   // 2. unwrap WBNB
157   WBNB.withdraw(_assets);
158
159   // 3. transfer BNB to receiver
160   (bool success, ) = receiver.call{ value: assets }("");
```

```
161   require(success, "transfer failed");
162 }
```

Listing 2.3: src/provider/BNBProvider.sol

Impact Users will lose their funds.

Suggestion Use the `_assets` variable instead of `assets`.

2.1.4 Lack of checks on the variable `_userLpRate` in the function `initialize()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `SlisBNBProvider` contract, the `initialize()` function sets the `userLpRate` variable based on the input `_userLpRate`. However, it does not check the input `_userLpRate` (i.e., `_userLpRate <= 1e18 && _userLpRate <= exchangeRate`), which may lead to an invalid value assignment for the variable `userLpRate`. As a result, LP tokens minted to delegatees might exceed the threshold calculated with the `exchangeRate`.

```
 95 function initialize(
 96     address admin,
 97     address manager,
 98     uint128 _userLpRate
 99 ) public initializer {
100   require(admin != address(0), "admin is the zero address");
101   require(manager != address(0), "manager is the zero address");
102
103   __AccessControl_init();
104
105   _grantRole(DEFAULT_ADMIN_ROLE, admin);
106   _grantRole(MANAGER, manager);
107
108   userLpRate = _userLpRate;
109 }
```

Listing 2.4: src/moolah/SlisBNBProvider.sol

```
316 function setUserLpRate(uint128 _userLpRate) external onlyRole(MANAGER) {
317   require(_userLpRate <= 1e18 && _userLpRate <= exchangeRate, "userLpRate invalid");
318
319   userLpRate = _userLpRate;
320   emit UserLpRateChanged(userLpRate);
321 }
```

Listing 2.5: src/moolah/SlisBNBProvider.sol

Impact LP tokens minted to delegatees might exceed the threshold calculated with the `exchangeRate`.

Suggestion Add checks on the `_userLpRate` variable in the `initialize()` function.

2.1.5 The `delegatee` can be maliciously reset

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `SlisBNBProvider` contract, the `delegateAllTo()` function allows users to set delegatees to hold their LP tokens. However, in the `supplyCollateral()` function, anyone can supply collateral on behalf of any users. Moreover, this function will modify users' `delegatee` to themself (i.e., `onBehalf`). As a result, this flaw design allows bad actors to supply tiny collateral on behalf of other users, preventing them from delegating their LP tokens.

```
112 function supplyCollateral(
113   MarketParams memory marketParams,
114   uint256 assets,
115   address onBehalf,
116   bytes calldata data
117 ) external {
118   require(assets > 0, "zero supply amount");
119   require(marketParams.collateralToken == token, "invalid collateral token");
120
121   // transfer token from user to this contract
122   IERC20(token).safeTransferFrom(msg.sender, address(this), assets);
123
124   // supply to Moolah
125   IERC20(token).safeIncreaseAllowance(address(MOOLAH), assets);
126   MOOLAH.supplyCollateral(marketParams, assets, onBehalf, data);
127
128
129   // get current delegatee
130   address oldDelegatee = delegation[onBehalf];
131   // burn all lpToken from old delegatee
132   if (oldDelegatee != onBehalf && oldDelegatee != address(0)) {
133     _safeBurnLp(oldDelegatee, userLp[onBehalf]);
134     // clear user's lpToken record
135     userLp[onBehalf] = 0;
136   }
137   // update delegatee
138   delegation[onBehalf] = onBehalf;
139
140   // rebalance user's lpToken
141   (,uint256 latestLpBalance) = _syncPosition(marketParams.id(), onBehalf);
142
143   emit Deposit(onBehalf, assets, latestLpBalance);
144 }
```

Listing 2.6: src/moolah/SlisBNBProvider.sol

```
270 function delegateAllTo(address newDelegatee)
271 external
272 {
273   require(
```

```
274     newDelegatee != address(0) &&
275     newDelegatee != delegation[msg.sender],
276     "newDelegatee cannot be zero address or same as current delegatee"
277   );
278   // current delegatee
279   address oldDelegatee = delegation[msg.sender];
280   // burn all lpToken from account or delegatee
281   _safeBurnLp(oldDelegatee, userLp[msg.sender]);
282   // update delegatee record
283   delegation[msg.sender] = newDelegatee;
284   // clear user's lpToken record
285   userLp[msg.sender] = 0;
286   // rebalance user's lpToken
287   _rebalanceUserLp(msg.sender);
288
289   emit ChangeDelegateTo(msg.sender, oldDelegatee, newDelegatee);
290 }
```

Listing 2.7: src/moolah/SlisBNBProvider.sol

Impact Users can be prevented from delegating their LP tokens.

Suggestion Revise the logic accordingly.

2.1.6 Incorrect calculation of the `wrapAmount` variable

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `repay()` function of the `BNBProvider` contract, users can repay their borrowed `WBNB` using `BNB`. Specifically, when users choose to repay by specifying the share amount, the function calculates the corresponding amount (i.e., the variable `wrapAmount`) of `WBNB` via the `toAssetUp()` function (line 186) and then approves the `wrapAmount` of `WBNB` to the `Moolah` contract. However, there is a lack of interest accrual before the calculation of the variable `wrapAmount`. As a result, the value of `wrapAmount` is smaller than the actual amount owed, causing the repayment to fail.

```
170 function repay(
171   MarketParams calldata marketParams,
172   uint256 assets,
173   uint256 shares,
174   address onBehalf,
175   bytes calldata data
176 ) external payable returns (uint256 _assets, uint256 _shares) {
177   require(UtilsLib.exactlyOneZero(assets, shares), ErrorsLib.INCONSISTENT_INPUT);
178   require(marketParams.loanToken == address(WBNB), "invalid loan token");
179   require(msg.value >= assets, "invalid BNB amount");
180
181   uint256 wrapAmount = assets;
182   if (wrapAmount == 0) {
183     // If assets is 0, we need to wrap the shares amount
```

```
184     require(shares > 0, ErrorsLib.ZERO_ASSETS);
185     Market memory market = MOOLAH.market(marketParams.id());
186     wrapAmount = shares.toAssetsUp(market.totalBorrowAssets, market.totalBorrowShares);
187   }
188
189   // 1. wrap BNB to WBNB
190   WBNB.deposit{ value: wrapAmount }();
191   // 2. approve moolah to transfer WBNB
192   require(WBNB.approve(address(MOOLAH), wrapAmount));
193   // 3. repay WBNB to moolah
194   (_assets, _shares) = MOOLAH.repay(marketParams, assets, shares, onBehalf, data);
195
196   // 4. return excess BNB to sender
197   if (msg.value > wrapAmount) {
198     (bool success, ) = msg.sender.call{ value: msg.value - wrapAmount }("");
199     require(success, "transfer failed");
200   }
201 }
```

Listing 2.8: src/provider/BNBProvider.sol

Impact Users' repayment will fail due to the incorrect calculation of the variable `wrapAmount`.

Suggestion Revise the logic accordingly.

2.1.7 Users can obtain LP tokens without supplying collateral

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Moolah` contract, the `MANAGER` role needs to invoke the `addProvider()` and `removeProvider()` functions to enable and disable providers (e.g., the `SlisBNBProvider` contract) for certain collaterals. After that, if the provider exists for collaterals, the `supplyCollateral()` and `withdrawCollateral()` functions of the `Moolah` contract restrict the `msg.sender` to corresponding providers. Moreover, in the `SlisBNBProvider` contract, users can earn LP tokens via supplying `slisBNB` tokens to the `Moolah` contract. However, this design is vulnerable to MEV attacks in two scenarios.

1. Bad actors can front-run the invocation of the `addProvider()` function by invoking the `Moolah.supplyCollateral()`, `SlisBNBProvider.syncUserLp()`, `Moolah.withdrawCollateral()` functions sequentially. As a result, the bad actor can obtain LP tokens without supplying any collateral..

2. In contrast, bad actors can invoke the `Moolah.withdrawCollateral()` function after the invocation of the `Moolah.removeProvider()` to withdraw their collateral without burning their LP tokens.

```
176 function addProvider(address token, address provider) external onlyRole(MANAGER) {
177   require(token != address(0), ErrorsLib.ZERO_ADDRESS);
178   require(provider != address(0), ErrorsLib.ZERO_ADDRESS);
179   require(providers[token] != provider, ErrorsLib.ALREADY_SET);
```

```
180
181   providers[token] = provider;
182
183   emit EventsLib.AddProvider(token, provider);
184 }
```

Listing 2.9: src/moolah/Moolah.sol

```
186 function removeProvider(address token) external onlyRole(MANAGER) {
187   require(token != address(0), ErrorsLib.ZERO_ADDRESS);
188   require(providers[token] != address(0), ErrorsLib.NOT_SET);
189
190   address provider = providers[token];
191
192   delete providers[token];
193
194   emit EventsLib.RemoveProvider(token, provider);
195 }
```

Listing 2.10: src/moolah/Moolah.sol

```
370 function supplyCollateral(
371   MarketParams memory marketParams,
372   uint256 assets,
373   address onBehalf,
374   bytes calldata data
375 ) external whenNotPaused nonReentrant {
376   Id id = marketParams.id();
377   require(market[id].lastUpdate != 0, ErrorsLib.MARKET_NOT_CREATED);
378   require(assets != 0, ErrorsLib.ZERO_ASSETS);
379   require(onBehalf != address(0), ErrorsLib.ZERO_ADDRESS);
380   if (providers[marketParams.collateralToken] != address(0)) {
381     require(msg.sender == providers[marketParams.collateralToken], ErrorsLib.NOT_PROVIDER);
382   }
383   // Don't accrue interest because it's not required and it saves gas.
384
385   position[id][onBehalf].collateral += assets.toUint128();
386
387   emit EventsLib.SupplyCollateral(id, msg.sender, onBehalf, assets);
388
389   if (data.length > 0) IMoolahSupplyCollateralCallback(msg.sender).onMoolahSupplyCollateral(
          assets, data);
390
391   IERC20(marketParams.collateralToken).safeTransferFrom(msg.sender, address(this), assets);
392 }
```

Listing 2.11: src/moolah/Moolah.sol

```
395 function withdrawCollateral(
396   MarketParams memory marketParams,
397   uint256 assets,
398   address onBehalf,
399   address receiver
```

```
400 ) external whenNotPaused nonReentrant {
401   Id id = marketParams.id();
402   require(market[id].lastUpdate != 0, ErrorsLib.MARKET_NOT_CREATED);
403   require(assets != 0, ErrorsLib.ZERO_ASSETS);
404   require(receiver != address(0), ErrorsLib.ZERO_ADDRESS);
405   // No need to verify that onBehalf != address(0) thanks to the following authorization check.
406   if (providers[marketParams.collateralToken] != address(0)) {
407     require(msg.sender == providers[marketParams.collateralToken] && receiver == providers[
          marketParams.collateralToken], ErrorsLib.NOT_PROVIDER);
408   } else {
409     require(_isSenderAuthorized(onBehalf), ErrorsLib.UNAUTHORIZED);
410   }
411
412   _accrueInterest(marketParams, id);
413
414   position[id][onBehalf].collateral -= assets.toUint128();
415
416   require(_isHealthy(marketParams, id, onBehalf), ErrorsLib.INSUFFICIENT_COLLATERAL);
417
418   emit EventsLib.WithdrawCollateral(id, msg.sender, onBehalf, receiver, assets);
419
420   IERC20(marketParams.collateralToken).safeTransfer(receiver, assets);
421 }
```

Listing 2.12: src/moolah/Moolah.sol

```
297 function syncUserLp(Id id, address _account) external {
298   (bool rebalanced,) = _syncPosition(id, _account);
299   require(rebalanced, "already synced");
300 }
```

Listing 2.13: src/moolah/SlisBNBProvider.sol

Impact Bad actors can obtain LP tokens without supplying collateral in the `Moolah` contract.

Suggestion Revise the logic accordingly.

2.1.8 Failure of supplying `WBNB` as collateral

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Moolah` contract, the `BNBProvider` contract is designated as the provider for `WBNB` collateral. However, this design prevents users from directly supplying `WBNB` as collateral in the `Moolah` contract.

```
370 function supplyCollateral(
371   MarketParams memory marketParams,
372   uint256 assets,
373   address onBehalf,
374   bytes calldata data
375 ) external whenNotPaused nonReentrant {
```

```
376   Id id = marketParams.id();
377   require(market[id].lastUpdate != 0, ErrorsLib.MARKET_NOT_CREATED);
378   require(assets != 0, ErrorsLib.ZERO_ASSETS);
379   require(onBehalf != address(0), ErrorsLib.ZERO_ADDRESS);
380   if (providers[marketParams.collateralToken] != address(0)) {
381     require(msg.sender == providers[marketParams.collateralToken], ErrorsLib.NOT_PROVIDER);
382   }
383   // Don't accrue interest because it's not required and it saves gas.
384
385   position[id][onBehalf].collateral += assets.toUint128();
386
387   emit EventsLib.SupplyCollateral(id, msg.sender, onBehalf, assets);
388
389   if (data.length > 0) IMoolahSupplyCollateralCallback(msg.sender).onMoolahSupplyCollateral(
          assets, data);
390
391   IERC20(marketParams.collateralToken).safeTransferFrom(msg.sender, address(this), assets);
392 }
```

Listing 2.14: src/moolah/Moolah.sol

```
395 function withdrawCollateral(
396   MarketParams memory marketParams,
397   uint256 assets,
398   address onBehalf,
399   address receiver
400 ) external whenNotPaused nonReentrant {
401   Id id = marketParams.id();
402   require(market[id].lastUpdate != 0, ErrorsLib.MARKET_NOT_CREATED);
403   require(assets != 0, ErrorsLib.ZERO_ASSETS);
404   require(receiver != address(0), ErrorsLib.ZERO_ADDRESS);
405   // No need to verify that onBehalf != address(0) thanks to the following authorization check.
406   if (providers[marketParams.collateralToken] != address(0)) {
407     require(msg.sender == providers[marketParams.collateralToken] && receiver == providers[
          marketParams.collateralToken], ErrorsLib.NOT_PROVIDER);
408   } else {
409     require(_isSenderAuthorized(onBehalf), ErrorsLib.UNAUTHORIZED);
410   }
411
412   _accrueInterest(marketParams, id);
413
414   position[id][onBehalf].collateral -= assets.toUint128();
415
416   require(_isHealthy(marketParams, id, onBehalf), ErrorsLib.INSUFFICIENT_COLLATERAL);
417
418   emit EventsLib.WithdrawCollateral(id, msg.sender, onBehalf, receiver, assets);
419
420   IERC20(marketParams.collateralToken).safeTransfer(receiver, assets);
421 }
```

Listing 2.15: src/moolah/Moolah.sol

Impact Users are prevented from directly supplying `WBNB` in the `Moolah` contract.

Suggestion Revise the logic accordingly.

2.1.9 Lack of checks on the variable `wrapAmount` in the `repay()` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `BNBProvider` contract, the `repay()` function does not check if the value is between the variable `wrapAmount` and msg.value. Without the check, the `repay()` function may fail due to insufficient funds.

```
170 function repay(
171   MarketParams calldata marketParams,
172   uint256 assets,
173   uint256 shares,
174   address onBehalf,
175   bytes calldata data
176 ) external payable returns (uint256 _assets, uint256 _shares) {
177   require(UtilsLib.exactlyOneZero(assets, shares), ErrorsLib.INCONSISTENT_INPUT);
178   require(marketParams.loanToken == address(WBNB), "invalid loan token");
179   require(msg.value >= assets, "invalid BNB amount");
180
181   uint256 wrapAmount = assets;
182   if (wrapAmount == 0) {
183     // If assets is 0, we need to wrap the shares amount
184     require(shares > 0, ErrorsLib.ZERO_ASSETS);
185     Market memory market = MOOLAH.market(marketParams.id());
186     wrapAmount = shares.toAssetsUp(market.totalBorrowAssets, market.totalBorrowShares);
187   }
188
189   // 1. wrap BNB to WBNB
190   WBNB.deposit{ value: wrapAmount }();
191   // 2. approve moolah to transfer WBNB
192   require(WBNB.approve(address(MOOLAH), wrapAmount));
193   // 3. repay WBNB to moolah
194   (_assets, _shares) = MOOLAH.repay(marketParams, assets, shares, onBehalf, data);
195
196   // 4. return excess BNB to sender
197   if (msg.value > wrapAmount) {
198     (bool success, ) = msg.sender.call{ value: msg.value - wrapAmount }("");
199     require(success, "transfer failed");
200   }
201 }
```

Listing 2.16: src/provider/BNBProvider.sol

Impact The `repay()` function may fail due to insufficient funds.

Suggestion Add a check for `msg.value`.

2.1.10 Potential DoS due to the lack of callback implementations

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `BNBProvider` contract, the `repay()` and `supplyCollateral()` functions allow users to trigger the callback mechanism in the `Moolah` contract based on the input `data`. However, there is no implementation of callback functions in the `BNBProvider` contract, leading the invocations to fail when the input `data` is not empty.

The `supplyCollateral()` function in the `SlisBNBProvider` contract has the same problem.

```
207 function supplyCollateral(
208   MarketParams calldata marketParams,
209   address onBehalf,
210   bytes calldata data
211 ) external payable {
212   uint256 assets = msg.value;
213   require(assets > 0, ErrorsLib.ZERO_ASSETS);
214   require(marketParams.collateralToken == address(WBNB), "invalid collateral token");
215
216   // 1. deposit WBNB
217   WBNB.deposit{ value: assets }();
218   // 2. approve moolah to transfer WBNB
219   require(WBNB.approve(address(MOOLAH), assets));
220   // 3. supply collateral to moolah
221   MOOLAH.supplyCollateral(marketParams, assets, onBehalf, data);
222 }
```

Listing 2.17: src/provider/BNBProvider.sol

```
112 function supplyCollateral(
113   MarketParams memory marketParams,
114   uint256 assets,
115   address onBehalf,
116   bytes calldata data
117 ) external {
118   require(assets > 0, "zero supply amount");
119   require(marketParams.collateralToken == token, "invalid collateral token");
120
121   // transfer token from user to this contract
122   IERC20(token).safeTransferFrom(msg.sender, address(this), assets);
123
124   // supply to Moolah
125   IERC20(token).safeIncreaseAllowance(address(MOOLAH), assets);
126   MOOLAH.supplyCollateral(marketParams, assets, onBehalf, data);
127
128
129   // get current delegatee
130   address oldDelegatee = delegation[onBehalf];
131   // burn all lpToken from old delegatee
132   if (oldDelegatee != onBehalf && oldDelegatee != address(0)) {
```

```
133     _safeBurnLp(oldDelegatee, userLp[onBehalf]);
134     // clear user's lpToken record
135     userLp[onBehalf] = 0;
136   }
137   // update delegatee
138   delegation[onBehalf] = onBehalf;
139
140   // rebalance user's lpToken
141   (,uint256 latestLpBalance) = _syncPosition(marketParams.id(), onBehalf);
142
143   emit Deposit(onBehalf, assets, latestLpBalance);
144 }
```

Listing 2.18: src/moolah/SlisBNBProvider.sol

```
170 function repay(
171   MarketParams calldata marketParams,
172   uint256 assets,
173   uint256 shares,
174   address onBehalf,
175   bytes calldata data
176 ) external payable returns (uint256 _assets, uint256 _shares) {
177   require(UtilsLib.exactlyOneZero(assets, shares), ErrorsLib.INCONSISTENT_INPUT);
178   require(marketParams.loanToken == address(WBNB), "invalid loan token");
179   require(msg.value >= assets, "invalid BNB amount");
180
181   uint256 wrapAmount = assets;
182   if (wrapAmount == 0) {
183     // If assets is 0, we need to wrap the shares amount
184     require(shares > 0, ErrorsLib.ZERO_ASSETS);
185     Market memory market = MOOLAH.market(marketParams.id());
186     wrapAmount = shares.toAssetsUp(market.totalBorrowAssets, market.totalBorrowShares);
187   }
188
189   // 1. wrap BNB to WBNB
190   WBNB.deposit{ value: wrapAmount }();
191   // 2. approve moolah to transfer WBNB
192   require(WBNB.approve(address(MOOLAH), wrapAmount));
193   // 3. repay WBNB to moolah
194   (_assets, _shares) = MOOLAH.repay(marketParams, assets, shares, onBehalf, data);
195
196   // 4. return excess BNB to sender
197   if (msg.value > wrapAmount) {
198     (bool success, ) = msg.sender.call{ value: msg.value - wrapAmount }("");
199     require(success, "transfer failed");
200   }
201 }
```

Listing 2.19: src/provider/BNBProvider.sol

Impact Functions may fail due to the lack of callback implementations.

Suggestion Revise the logic accordingly.

Feedback from the project The project believes that this is a known issue and there are no plans to support the callback mechanism so far.

2.2 Recommendation

2.2.1 Lack of invoking function `_disableInitializers()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contracts `BNBProvider` and `SlisBNBProvider`, the function `_disableInitializers()` is not invoked in the constructor. Invoking this function prevents the contract itself from being initialized, thereby avoiding unexpected behaviors.

```
73  constructor(
74    address moolah,
75    address _token,
76    address _stakeManager,
77    address _lpToken
78  ) {
79    require(moolah != address(0), "moolah is the zero address");
80    require(_token != address(0), "token is the zero address");
81    require(_stakeManager != address(0), "stakeManager is the zero address");
82    require(_lpToken != address(0), "lpToken is the zero address");
83
84    MOOLAH = IMoolah(moolah);
85    token = _token;
86    stakeManager = IStakeManager(_stakeManager);
87    lpToken = ILpToken(_lpToken);
88  }
```

Listing 2.20: src/moolah/SlisBNBProvider.sol

```
45  constructor(address moolah, address moolahVault, address wbnb) {
46    require(moolah != address(0), ErrorsLib.ZERO_ADDRESS);
47    require(moolahVault != address(0), ErrorsLib.ZERO_ADDRESS);
48    require(moolah == address(IMoolahVault(moolahVault).MOOLAH()), ErrorsLib.NOT_SET);
49    require(wbnb != address(0), ErrorsLib.ZERO_ADDRESS);
50    require(wbnb == IMoolahVault(moolahVault).asset(), "asset mismatch");
51
52    MOOLAH = IMoolah(moolah);
53    MOOLAH_VAULT = IMoolahVault(moolahVault);
54    WBNB = IWBNB(wbnb);
55  }
```

Listing 2.21: src/provider/BNBProvider.sol

Suggestion Invoke the function `_disableInitializers()` in the constructor.

2.2.2 Lack of zero address checks

Status Fixed in `Version 2`

Introduced by Version 1

Description In function `withdraw()`, `redeem()`, `borrow()`, and `withdrawCollateral()`, the address variable (i.e., `receiver`) is not checked to ensure it is not zero. It is recommended to add such checks to prevent potential mis-operations.

```
104 function withdraw(uint256 assets, address payable receiver, address owner) external returns (
        uint256 shares) {
105   require(assets > 0, ErrorsLib.ZERO_ASSETS);
106
107   // 1. withdraw WBNB from moolah vault
108   shares = MOOLAH_VAULT.withdrawFor(assets, owner, msg.sender);
109
110   // 2. unwrap WBNB
111   WBNB.withdraw(assets);
112
113   // 3. transfer WBNB to receiver
114   (bool success, ) = receiver.call{ value: assets }("");
115   require(success, "transfer failed");
116 }
```

Listing 2.22: src/provider/BNBProvider.sol

```
122 function redeem(uint256 shares, address payable receiver, address owner) external returns (
        uint256 assets) {
123   require(shares > 0, ErrorsLib.ZERO_ASSETS);
124
125   // 1. redeem WBNB from moolah vault
126   assets = MOOLAH_VAULT.redeemFor(shares, owner, msg.sender);
127
128   // 2. unwrap WBNB
129   WBNB.withdraw(assets);
130
131   // 3. transfer BNB to receiver
132   (bool success, ) = receiver.call{ value: assets }("");
133   require(success, "transfer failed");
134 }
```

Listing 2.23: src/provider/BNBProvider.sol

```
142 function borrow(
143   MarketParams calldata marketParams,
144   uint256 assets,
145   uint256 shares,
146   address onBehalf,
147   address payable receiver
148 ) external returns (uint256 _assets, uint256 _shares) {
149   // No need to verify assets and shares, as they are already verified in the Moolah contract.
150   require(marketParams.loanToken == address(WBNB), "invalid loan token");
151   require(isSenderAuthorized(msg.sender, onBehalf), ErrorsLib.UNAUTHORIZED);
152
153   // 1. borrow WBNB from moolah
154   (_assets, _shares) = MOOLAH.borrow(marketParams, assets, shares, onBehalf, address(this));
155
```

```
156   // 2. unwrap WBNB
157   WBNB.withdraw(_assets);
158
159   // 3. transfer BNB to receiver
160   (bool success, ) = receiver.call{ value: assets }("");
161   require(success, "transfer failed");
162 }
```

Listing 2.24: src/provider/BNBProvider.sol

```
229 function withdrawCollateral(
230   MarketParams calldata marketParams,
231   uint256 assets,
232   address onBehalf,
233   address payable receiver
234 ) external {
235   require(marketParams.collateralToken == address(WBNB), "invalid collateral token");
236   require(isSenderAuthorized(msg.sender, onBehalf), ErrorsLib.UNAUTHORIZED);
237
238   // 1. withdraw WBNB from moolah by specifying the amount
239   MOOLAH.withdrawCollateral(marketParams, assets, onBehalf, address(this));
240
241   // 2. unwrap WBNB
242   WBNB.withdraw(assets);
243
244   // 3. transfer BNB to receiver
245   (bool success, ) = receiver.call{ value: assets }("");
246   require(success, "transfer failed");
247 }
```

Listing 2.25: src/provider/BNBProvider.sol

Suggestion Add non-zero address checks accordingly.

2.2.3 Add state change checks in the `setUserLpRare()` function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the protocol, the `MANAGER` role can manage `userLpRate` through the `setUserLpRate()` function. It is recommended to implement state change checks on the current status of `userLpRate` to ensure that it differs from the newly updated value.

```
316 function setUserLpRate(uint128 _userLpRate) external onlyRole(MANAGER) {
317   require(_userLpRate <= 1e18 && _userLpRate <= exchangeRate, "userLpRate invalid");
318
319   userLpRate = _userLpRate;
320   emit UserLpRateChanged(userLpRate);
321 }
```

Listing 2.26: src/moolah/SlisBNBProvider.sol

Suggestion Add a state change check for the current status of `userLpRate` in the function `setUserLpRate()`.

2.2.4 Unify the the use of the variable `RATE_DENOMINATOR`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `setUserLpRate()` function, it uses `1e18` to validate the `_userLpRate` variable. However there is also a constant `RATE_DENOMINATOR` that is equal to `1e18`. It is recommended to unify the use of the `RATE_DENOMINATOR` variable.

```
316 function setUserLpRate(uint128 _userLpRate) external onlyRole(MANAGER) {
317   require(_userLpRate <= 1e18 && _userLpRate <= exchangeRate, "userLpRate invalid");
318
319   userLpRate = _userLpRate;
320   emit UserLpRateChanged(userLpRate);
321 }
```

Listing 2.27: src/moolah/SlisBNBProvider.sol

Suggestion In the `setUserLpRate()` function, use `RATE_DENOMINATOR` instead of `1e18`.

2.2.5 Add rescue token functionality in `SlisBNBProvider` and `BNBProvider` contracts

Status Confirmed

Introduced by `Version 1`

Description Neither of the `BNBProvider` and `SlisBNBProvider` contracts has a function to rescue native BNB or ERC20 tokens, which might be mistakenly sent to the contracts. It is recommended to add a rescue token function in both the `BNBProvider` and `SlisBNBProvider` contracts.

Suggestion Add rescue token functionality in the `BNBProvider` and `SlisBNBProvider` contracts.

2.3 Note

2.3.1 The minting and burning mechanism of LP tokens

Introduced by `Version 1`

Description In the contract `SlisBNBProvider`, users can mint LP tokens to their delegatees through the `delegateAllTo()` function and restore the LP tokens through the `syncUserLp()` function. It is important to note that a delegatee's LP tokens should only be burned or minted by the users who delegate to him in both `SlisBNBProvider` and other external contracts. Otherwise, if a delegatee's LP tokens are capable of being burned without decreasing the user's collateral, the user can restore the LP tokens through the `delegateAllTo()` function to a new delegatee, which could potentially result in a double-spending risk.

2.3.2 Potential centralization risks

Introduced by Version 1

Description In this protocol, several privileged roles (e.g., the `provider` role) can conduct sensitive operations, which introduces potential centralization risks. For example, in the `Moolah` contract, if `providers[marketParams.loanToken]` is set to be a malicious address, then it can borrow funds by using anyone's collateral through the `borrow()` function. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.3 Sufficient capacity in the MPC wallets

Introduced by Version 1

Description In the `SlisBNBProvider` contract, the `_mintToMPCs()` function mints LP tokens for the MPC wallets, with each wallet's minting limit being defined by `wallet.cap`. If the LP tokens to be minted exceed the current `wallet's cap`, the excess LP tokens are redistributed to other wallets. It is important to ensure that the total MPC wallets capacity is sufficient to avoid a potential DoS risk.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS