Competitive Security Assessment

Bagful_io

Oct 5th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3′s Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.
The comprehensive examination and auditing scope includes:
• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.
The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

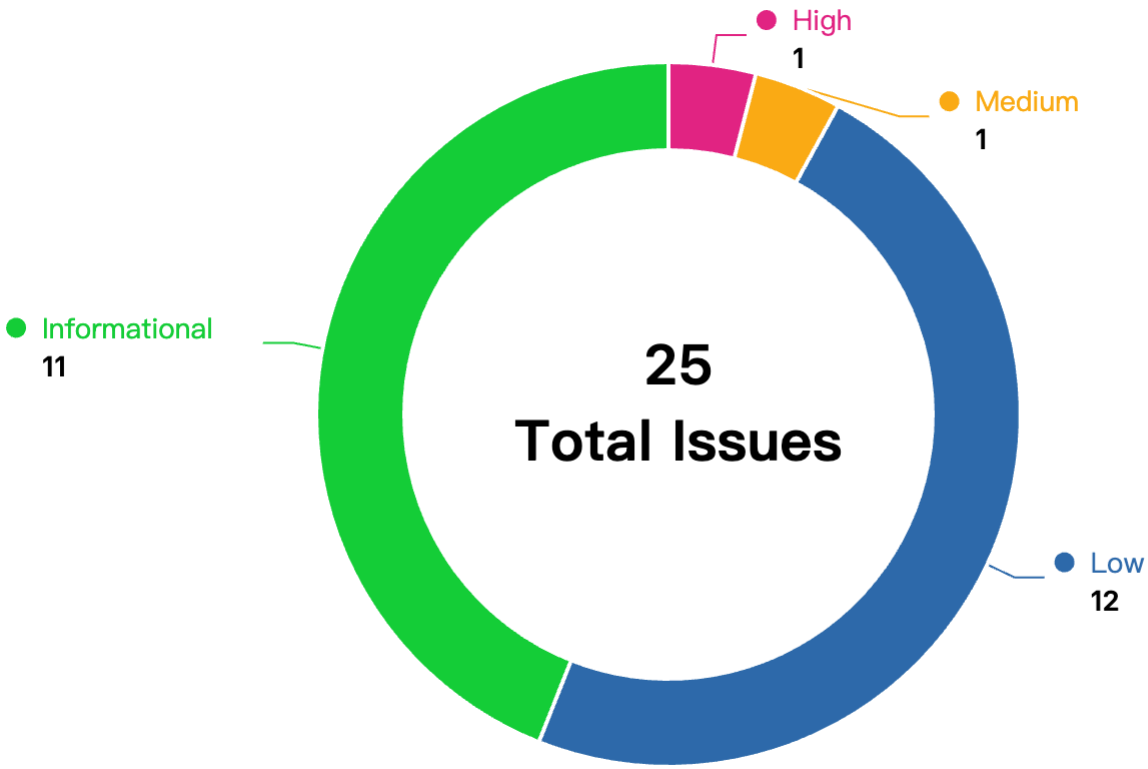
Overview

Project Name	Bagful_io
Language	solidity
Codebase	• https://github.com/bagfulcrew/contracts/tree/main • audit version-748ebdc7a77b5f0b7fcbb09f3e8aa5a5391dae30 • final version-76afb2e23eecc07a3c74162392d8d94166c12861

Audit Scope

File	SHA256 Hash
farms/mendiCompoundFarm.sol	534e726b7a1aa959409895cd0a559e93f1973cde4a9ca63908f6f480d4abccff
comm/TransferHelper.sol	6f6a3008228906279e0e297bb8352b257f33942c79189ce9c8367074d7b67621

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
BIO-1	The `deposit` and `withdraw` functions can potentially mess up because dependency of `mendiCToken` is not fully compatible	Logical	High	Fixed	***
BIO-2	User is given more CToken than it recieves at Mendi	Logical	Medium	Fixed	***
BIO-3	`withdraw` function may revert after a new reward token is added into the pool	Logical	Low	Fixed	***
BIO-4	Use `_disableInitializers()` to prevent front-running on the initialize function	Language Specific	Low	Fixed	***
BIO-5	The functions of `cTokenToUnderlying` and `underlyingToCToken` are possibly using stale `exchangeRate` data	Logical	Low	Acknowledged	***

BIO-6	The function `balanceOfUnderlying()` has wrong code implementation so it cannot return the correct result	Logical	Low	Fixed	***
BIO-7	The `harvest` function forgot to `updatePool()` when `_extraReward.isSettledIncome() == false`	Logical	Low	Fixed	***
BIO-8	Potential inconsistency when handling reward distribution	Logical	Low	Fixed	***
BIO-9	Potential for Reward Token Drain During Removal	Integer Overflow and Underflow	Low	Fixed	***
BIO-10	Potential DOS risk in the function `distributeAllRewards` and `removeExtraReward`	Logical	Low	Fixed	***
BIO-11	Not calling approve(0) before setting a new approval causes the call to revert when used with Tether (USDT)	Logical	Low	Fixed	***
BIO-12	Lack of Time-Based Restrictions on Reward Distribution	Flash Loan Attacks	Low	Acknowledged	***
BIO-13	In the event of default happening on the Mendi's side, the farming contract will allow the fast runners escape with no loss and slow hands bear all the loss	Logical	Low	Acknowledged	***
BIO-14	Arbitrary Timestamp Setting	Privilege Related	Low	Fixed	***
BIO-15	`startMining` Function Lacks Event Emission	Logical	Informational	Fixed	***
BIO-16	Unused Libs	Gas Optimization	Informational	Fixed	***
BIO-17	The initialize function lacks zero-address checks	Logical	Informational	Fixed	***

BIO-18	Replace `abi.encodeWithSelector` with `abi.encodeCall` which keeps the code typo/type safe	Language Specific	Informational	Fixed	***
BIO-19	Redundant approve() Call in the deposit() Function	Logical	Informational	Fixed	***
BIO-20	Ownership change should use two-step process	Privilege Related	Informational	Acknowledged	***
BIO-21	Missing __gap variable in the contract BagfulMendiCompoundFarm	DOS	Informational	Fixed	***
BIO-22	Lack of Pause Functionality	Privilege Related	Informational	Fixed	***
BIO-23	Initializing totalDeposits to 0 is unnecessary in the context of the `initialize` function	Code Style	Informational	Fixed	***
BIO-24	Incompatibility With Deflationary Tokens	Logical	Informational	Fixed	***
BIO-25	An Array's length Should Be Cached To Save Gas in Loops	Gas Optimization	Informational	Fixed	***

BIO-1:The `deposit` and `withdraw` functions can potentially mess up because dependency of `mendiCToken` is not fully compatible

Category	Severity	Client Response	Contributor
Logical	High	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L171

```
171: function withdraw(uint256 _amount) external nonReentrant {
```

Description

***: The withdraw() function of the BagfulMendiCompoundFarm contract does not correctly account for the possibility of incomplete redemptions from the cToken contract, leading to discrepancies between the expected and actual amount of assets redeemed. This can result in users potentially withdrawing more or fewer assets than they are entitled to, which could cause a critical vulnerability where the protocol loses funds or users can exploit it for financial gain.
When users call the withdraw() function, the contract interacts with the mendiCToken.redeemUnderlying(_amount) function, which is supposed to convert the underlying asset back from cTokens. However, the code does not verify whether the redemption process is successful or if the full _amount was redeemed.
Consider a scenario where the redeemUnderlying() call fails silently, or returns fewer tokens than _amount. For instance:

```
mendiCToken.redeemUnderlying(_amount); // Fails or redeems less than _amount

uint256 reduceCTokenAmount = underlyingToCToken(_amount);
if (userInfo.cTokenAmount > reduceCTokenAmount) {
    userInfo.cTokenAmount -= reduceCTokenAmount;
} else {
    userInfo.cTokenAmount = 0;
}

userInfo.underlyingAmount -= _amount;
```

Recommendation

***: To mitigate this issue, implement proper checks for the success of the redeemUnderlying() function and ensure that the exact amount of tokens is redeemed before proceeding with balance updates.
Check Redemption Success: Verify the return value of the redeemUnderlying() function and handle failures or partial redemptions appropriately.

```
uint256 redeemedAmount = mendiCToken.redeemUnderlying(_amount);
require(redeemedAmount == _amount, "Redemption failed or incomplete");
```

Adjust User Balances Based on Actual Redemption: Only reduce the user's underlyingAmount and cTokenAmount after confirming the actual amount redeemed.

```
userInfo.underlyingAmount -= redeemedAmount;
userInfo.cTokenAmount -= underlyingToCToken(redeemedAmount);
```

Fail Gracefully: If the redemption fails or is incomplete, revert the transaction and notify the user, ensuring the protocol remains secure and balances remain accurate.
By implementing these checks, you can prevent users from withdrawing more than they are entitled to, protect the protocol from losing funds, and ensure that the system remains trustworthy and accurate.

Client Response

client response : Fixed. The returned result of redeemUnderlying is determined, and can only continue if the deposit/withdraw is successfully proceeded.

BIO-2:User is given more CToken than it recieves at Mendi

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L155

```
155: uint256 calcCToken = underlyingToCToken(_amount);
```

Description

***: User is assigned more Ctoken amount than it actually gets at Mendi, this is becuase the exchnage rate is called at the wrong time.
Mendi is a Compound v2 fork, and exchnange rate is called before the effect of the deposited asset like so https://github.com/compound-finance/compound-protocol/blob/a3214f67b73310d547e00fc578e8355911c9d376/contracts/CToken.sol#L410C1-L425C1

```
        Exp memory exchangeRate = Exp({mantissa: exchangeRateStoredInternal()});

        ////////////////////////
        // EFFECTS & INTERACTIONS
        // (No safe failures beyond this point)

        /*
         *  We call `doTransferIn` for the minter and the mintAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  `doTransferIn` reverts if anything goes wrong, since we can't be sure if
         *  side-effects occurred. The function returns the amount actually transferred,
         *  in case of a fee. On success, the cToken holds an additional `actualMintAmount`
         *  of cash.
         */
        uint actualMintAmount = doTransferIn(minter, mintAmount);

        /*
         * We get the current exchange rate and calculate the number of cTokens to be minted:
         *  mintTokens = actualMintAmount / exchangeRate
         */

        uint mintTokens = div_(actualMintAmount, exchangeRate);
```

But mendiCompoundfarm calls it after

```
        assetToken.approve(address(mendiCToken), _amount); // approve twice @audit
        mendiCToken.mint(_amount);

        // Calculate the cToken
        uint256 calcCToken = underlyingToCToken(_amount);
        // so this essentailly returns the value of ctoken at mint.
        userInfo.underlyingAmount += _amount;
```

```
    /// @notice Calculate the underlying amount
     function underlyingToCToken(uint256 _underlyingAmount) public view returns (uint256) {
        uint256 exchangeRate = mendiCToken.exchangeRateStored();
        return (_underlyingAmount * 1e18) / exchangeRate;
     }
```

meaning it actually records more ctoken amount than the user actually gets minted at Mendi, cause each deposit increases the exchnage rate.

Recommendation

***: Call the exchang rate before minting in mendy like so

```
        uint256 calcCToken = underlyingToCToken(_amount);

        assetToken.approve(address(mendiCToken), _amount); // approve twice @audit
        mendiCToken.mint(_amount);
        // so this essentailly returns the value of ctoken at mint.
        userInfo.underlyingAmount += _amount;
```

Client Response

client response : Fixed.

BIO-3: `withdraw` function may revert after a new reward token is added into the pool

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L80-L93
- code/farms/mendiCompoundFarm.sol#L124-L205
- code/farms/mendiCompoundFarm.sol#L297-L307

```
80: /// @notice Add new reward token to pool
81:     /// @param _rewardTokenAddr The new reward token
82:     function addExtraReward(address _rewardTokenAddr) external onlyOwner {
83:         require(_rewardTokenAddr != address(0), "Invalid reward address");
84:
85:         for (uint256 i = 0; i < extraRewards.length; i++) {
86:             if (address(extraRewards[i]) == _rewardTokenAddr) {
87:                 revert RewardTokenExisted(_rewardTokenAddr);
88:             }
89:         }
90:
91:         extraRewards.push(IRewardNew(_rewardTokenAddr));
92:         emit AddExtraRewardToken(_rewardTokenAddr);
93:     }
```

```
124: /// @notice Deposit assets to the farm
125:     /// @param _amount The amount of assets to deposit
126:     function deposit(uint256 _amount) external payable nonReentrant {
127:         require(startTimestamp > 0, "Farm: mining not start!!");
128:
129:         UserInfo storage userInfo = userInfoMap[msg.sender];
130:
131:         // Distribute rewards to user
132:         distributeAllRewards(msg.sender);
133:
134:         // process WETH
135:         if (address(assetToken) == ethAddr) {
136:             require(_amount == 0, "Deposit invalid token");
137:
138:             if (msg.value > 0) {
139:                 _amount = _amount + msg.value;
140:             }
141:         } else {
142:             require(msg.value == 0, "Deposit invalid token");
143:             if (_amount > 0) {
```

```
144:                 TransferHelper.safeTransferFrom(address(assetToken), address(msg.sender), addre
ss(this), _amount);
145:             }
146:
147:             assetToken.approve(address(mendiCToken), _amount);
148:         }
149:
150:         // Save assets to Mendi
151:         assetToken.approve(address(mendiCToken), _amount);
152:         mendiCToken.mint(_amount);
153:
154:         // Calculate the cToken
155:         uint256 calcCToken = underlyingToCToken(_amount);
156:         userInfo.underlyingAmount += _amount;
157:         userInfo.cTokenAmount += calcCToken;
158:         userInfo.lastDepositTime = block.timestamp;
159:
160:         userAddrList.add(msg.sender);
161:
162:         totalDeposits += _amount;
163:
```

```
164:         updateAllRewards(msg.sender, _amount, true);
165:
166:         emit Deposit(msg.sender, _amount, calcCToken);
167:     }
168:
169:     /// @notice Withdraw assets from the farm
170:     /// @param _amount The amount of assets to withdraw
171:     function withdraw(uint256 _amount) external nonReentrant {
172:         require(_amount > 0, "Invalid deposit amount");
173:         require(startTimestamp > 0, "Mining not start!!");
174:
175:         UserInfo storage userInfo = userInfoMap[msg.sender];
176:         require(userInfo.underlyingAmount >= _amount, "Insufficient balance");
177:
178:         // Distribute rewards to user
179:         distributeAllRewards(msg.sender);
180:
181:         mendiCToken.redeemUnderlying(_amount);
182:
183:         uint256 reduceCTokenAmount = underlyingToCToken(_amount);
```

```
184:
185:         if (userInfo.cTokenAmount > reduceCTokenAmount) {
186:             userInfo.cTokenAmount -= reduceCTokenAmount;
187:         } else {
188:             userInfo.cTokenAmount = 0;
189:         }
190:
191:         userInfo.underlyingAmount -= _amount;
192:         userInfo.lastDepositTime = block.timestamp;
193:
194:         totalDeposits -= _amount;
195:
196:         if (address(assetToken) == ethAddr) {
197:             TransferHelper.safeTransferETH(msg.sender, _amount);
198:         } else {
199:             TransferHelper.safeTransfer(address(assetToken), msg.sender, _amount);
200:         }
201:
202:         updateAllRewards(msg.sender, _amount, false);
203:
```

```
204:         emit Withdraw(msg.sender, _amount);
205:     }
```

```
297: /// @notice Update all of rewards state
298:     /// @param _user The user address
299:     /// @param _amount The amount of assets
300:     /// @param depositFlag The deposit flag
301:     function updateAllRewards(address _user, uint256 _amount, bool depositFlag) internal {
302:         for (uint256 i = 0; i < extraRewards.length; i++) {
303:             if (!extraRewards[i].isRetired()) {
304:                 extraRewards[i].updateUserState(_user, _amount, depositFlag);
305:             }
306:         }
307:     }
```

Description

***: The function **`deposit`** and **`withdraw`** both call **`updateAllRewards`** function:

```
// in deposit function
updateAllRewards(msg.sender, _amount, true);
```

```
// in withdraw function
updateAllRewards(msg.sender, _amount, false);
```

The **`updateAllRewards`** function will update all of rewards state:

```
/// @notice Update all of rewards state
    /// @param _user The user address
    /// @param _amount The amount of assets
    /// @param depositFlag The deposit flag
    function updateAllRewards(address _user, uint256 _amount, bool depositFlag) internal {
        for (uint256 i = 0; i < extraRewards.length; i++) {
            if (!extraRewards[i].isRetired()) {
                extraRewards[i].updateUserState(_user, _amount, depositFlag);
            }
        }
    }
```

As per the MockReward, the function `updateUserState` will increase or decrease the user's balance based on `depositFlag`:

```
function updateUserState(address user, uint256 amount, bool deposit) external override onlyPool {
        if (!retired) {
            if (deposit) {
                userRewards[user].depositAmount += amount;
            } else {
                userRewards[user].depositAmount -= amount;
            }
        }
    }
```

The `MockReward` here is for testing purposes, if the logic of the official reward contract is similar to it, then it may cause the user to be unable to withdraw under certain conditions. Here is an example:

1. Bob calls `deposit` function with `_amount = 100`, the `deposit` function will call `updateAllRewards` and each reward contract will increase Bob's balance to 100
2. The owner adds a new reward contract into `extraRewards`, Bob's balance in this reward contract is 0 now
3. Bob calls `withdraw` function with `_amount = 100`, the `withdraw` function will call `updateAllRewards` and each reward contract will decrease Bob's balance. However, Bob's balance in the new reward contract is 0 now, that means when calling the new reward contract's `updateUserState`, it will revert due to math underflow.

Recommendation

***: Consider adding a check in `updateAllRewards`:

```
function updateAllRewards(address _user, uint256 _amount, bool depositFlag) internal {
        for (uint256 i = 0; i < extraRewards.length; i++) {
            if (!extraRewards[i].isRetired()) {
                if(depositFlag){
                   extraRewards[i].updateUserState(_user, _amount, depositFlag);
                }else{
                 UserRewardInfo memory userInfo = extraRewards[i].getUserRewardInfo(_user);
                 if(userInfo.depositAmount > 0){
                  extraRewards[i].updateUserState(_user, _amount, depositFlag);
                 }
                }

            }
        }
    }
```

Client Response

client response : Fixed.

BIO-4:Use `_disableInitializers()` to prevent front-running on the initialize function

Category	Severity	Client Response	Contributor
Language Specific	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L65-L78

```
65: function initialize(
66:         address _assets,
67:         address _mendiCToken,
68:         address _ethAddr
69:     ) public initializer {
70:         __Ownable_init(msg.sender);
71:         __ReentrancyGuard_init();
72:
73:         assetToken = IERC20(_assets);
74:         mendiCToken = IMendiCToken(_mendiCToken);
75:         ethAddr = _ethAddr;
76:
77:         totalDeposits = 0;
78:     }
```

Description

***: According to OZ'S guideline for protection of initialize function with `_disableInitializers()` method, implementation contracts should not remain uninitialized. Uninitialized contract can lead to attack where a malicious attacker can take over control of the contract.
Here is the link to the respective communication with OpenZeppelin staff:
https://forum.openzeppelin.com/t/what-does-disableinitializers-function-mean/28730/7
The contact `BagfulMendiCompoundFarm.sol` is initialized by `initialize` function:

```
function initialize(
        address _assets,
        address _mendiCToken,
        address _ethAddr
    ) public initializer {
        __Ownable_init(msg.sender);
        __ReentrancyGuard_init();

        assetToken = IERC20(_assets);
        mendiCToken = IMendiCToken(_mendiCToken);
        ethAddr = _ethAddr;

        totalDeposits = 0;
    }
```

Ensure prevention of initialization by an attacker which will have a direct impact on the contract as the implementation contract's constructor should have `_disableInitializers()` method.

POC

1. Proxy & Implementation contracts are deployed.
2. The Proxy delegates calls to `initialize()` which sets the owner and switches initialized to true in the state of the Proxy.
3. The storage of Implementation however is still intact.
4. An attacker calls `initialize()` directly on Implementation and sets himself as the owner.
5. From here, he has full control to perform any malicious activities.

Recommendation

***:** Invoke `_disableInitializers` in constructor.

```
+   constructor {
+       _disableInitializers()
+   }
    //...
```

Client Response

client response : Fixed.

BIO-5:The functions of `cTokenToUnderlying` and `underlyingToCToken` are possibly using stale `exchangeRate` data

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L341-L351

```
341: /// @notice Calculate the cToken amount
342:     function cTokenToUnderlying(uint256 _cTokenAmount) public view returns (uint256) {
343:         uint256 exchangeRate = mendiCToken.exchangeRateStored();
344:         return (_cTokenAmount * exchangeRate) / 1e18;
345:     }
346:
347:     /// @notice Calculate the underlying amount
348:     function underlyingToCToken(uint256 _underlyingAmount) public view returns (uint256) {
349:         uint256 exchangeRate = mendiCToken.exchangeRateStored();
350:         return (_underlyingAmount * 1e18) / exchangeRate;
351:     }
```

Description

***: The functions of **`cTokenToUnderlying`** and **`underlyingToCToken`** are querying the saved **`exchangeRate`** directly and using it; however, it does not account for the interest accured on the Mendi's side since the timestamp when the value of **`exchangeRate`** was most recently saved until now. This interest accumulation will change the effective **`exchangeRate`** at the moment, so it will be different from the saved **`exchangeRate`** value.
Actually, the Mendi's contract as an upstream dependency, has clearly shown the difference:

```
    function exchangeRateCurrent()
        public
        override
        nonReentrant
        returns (uint256)
    {
        accrueInterest();
        return exchangeRateStored();
    }

    /**
     * @notice Calculates the exchange rate from the underlying to the CToken
     * @dev This function does not accrue interest before calculating the exchange rate
     * @return Calculated exchange rate scaled by 1e18
     */
    function exchangeRateStored() public view override returns (uint256) {
        return exchangeRateStoredInternal();
    }
```

As we can see, the only and main difference is the code `accrueInterest()` needs to happen before querying the `exchangeRateStored()`.
This issue is only **LOW** severity, because in the current implementation where these two functions are used by other state mutating functions, they are used right after external calls like `mendiCToken.mint(_amount)` or `mendiCToken.redeemUnderlying(_amount)`. These calls to the Mendi's protocol will also lazy trigger the `accrueInterest()`. Therefore, inside the same transaction at the same timestamp it's not really needed to trigger the `accrueInterest()` again. As a result, although the method is not ideal, the current usecase is having minimal impact. That being said, when these two functions are used in other cases, there is still high probability that they are using stale data which can lead to wrong results of **MEDIUM** or **HIGH** impact.

Recommendation

***: Can create these two new functions accordingly which are guaranteed to be using non-stale `exchangeRate`:

```
function cTokenToUnderlyingCurrent(uint256 _cTokenAmount) public view returns (uint256) {
    uint256 exchangeRate = mendiCToken.exchangeRateCurrent();
    return (_cTokenAmount * exchangeRate) / 1e18;
}


function underlyingToCTokenCurrent(uint256 _underlyingAmount) public view returns (uint256) {
    uint256 exchangeRate = mendiCToken.exchangeRateCurrent();
    return (_underlyingAmount * 1e18) / exchangeRate;
}
```

Client Response

client response : Acknowledged. Mendi didn't provide the exchangeRateCurrent method.

BIO-6:The function `balanceOfUnderlying()` has wrong code implementation so it cannot return the correct result

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L243-L246

```
243: /// @notice Return the user underlying assets balance
244:     function balanceOfUnderlying() external view returns (uint256) {
245:         return assetToken.balanceOf(address(this));
246:     }
```

Description

***: The current implementation of the **`balanceOfUnderlying()`** is mainly about this line:

```
        return assetToken.balanceOf(address(this));
```

That line is wrong, because at the moment when a user deposited the underlying asset into the farming contract, the asset immediately got transferred into the **`mendiCToken`** contract by calling its **`mint`** function. Therefore, the **`address(this)`** no longer holds the **`assetToken`**, instead, it holds the corresponding **`cToken`** minted for it. As a result, the **`assetToken.balanceOf(address(this))`** is NOT the correct way to get the expected data, and actually, this call simply just returns **`0`** for most of the time.

Recommendation

***: To get the current amount of the underlying asset belonging to **`address(this)`**, the correct way should be utilizing its current balance of the **`mendiCToken`** and the **`exchangeRate`**, so that the amount of the **`assetToken`** can be calculated.
For example:

```
    function balanceOfUnderlying() external view returns (uint256) {
        uint256 cTokenAmount = mendiCToken.balanceOf(address(this));
        return cTokenToUnderlying(cTokenAmount);
    }
```

In addition, The dev commented this for the **`balanceOfUnderlying()`**:

```
    /// @notice Return the user underlying assets balance
```

The function above is only meant to get the amount of ALL users' asset aggregated in the farming contract. If the dev is planning to have a function to return the amount of underlying asset invested for an individual user, then should make another similar function, taking the input of **`_userAddress`** and utilize **`userInfoMap[_userAddress]`** to get the **`mendiCToken`** amount belonging to that particular user and then do the calculation.

Client Response

client response : Fixed.

BIO-7:The `harvest` function forgot to `updatePool()` when `_extraReward.isSettledIncome() == false`

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L209-L226

```
209: function harvest(address _user) external nonReentrant {
210:         require(startTimestamp > 0, "Mining not start!!");
211:         require(_user != address(0), "Farm: invalid user address");
212:
213:         UserInfo storage userInfo = userInfoMap[_user];
214:         require(userInfo.underlyingAmount > 0, "No deposit");
215:
216:         for (uint256 i = 0; i < extraRewards.length; i++) {
217:             IRewardNew _extraReward = extraRewards[i];
218:
219:             uint256 pendingRewards = extraRewards[i].calculateReward(_user,
220:                 _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);
221:
222:             if (pendingRewards > 0) {
223:                 _extraReward.distributeReward(_user, pendingRewards);
224:             }
225:         }
226:     }
```

Description

***: The functions **`distributeAllRewards`** and **`harvest`** are designed to have the same logic about distributing the accumulated extra rewards to the user. The only difference can be considered that, the **`harvest`** is an external function meant to proactively trigger the reward distribution, while **`distributeAllRewards`** is an internal function meant to be lazily triggered when users deposit or withdraw. Other than this, the rest of the logic is supposed to be the same for the two function.
However, we can see a major discrepancy which appear to be unintentional. See below:
`harvest`:

```
    function harvest(address _user) external nonReentrant {
        require(startTimestamp > 0, "Mining not start!!");
        require(_user != address(0), "Farm: invalid user address");

        UserInfo storage userInfo = userInfoMap[_user];
        require(userInfo.underlyingAmount > 0, "No deposit");

        for (uint256 i = 0; i < extraRewards.length; i++) {
            IRewardNew _extraReward = extraRewards[i];

            uint256 pendingRewards = extraRewards[i].calculateReward(_user,
                _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);

            if (pendingRewards > 0) {
                _extraReward.distributeReward(_user, pendingRewards);
            }
        }
    }
```

`distributeAllRewards`:

```
    function distributeAllRewards(address _user) internal {
        UserInfo storage userInfo = userInfoMap[_user];

        for (uint256 i = 0; i < extraRewards.length; i++) {
            IRewardNew _extraReward = extraRewards[i];

            uint256 rewardAmount = extraRewards[i].calculateReward(_user,
                _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);

            // Liquidity reward
            if (_extraReward.isSettledIncome() == false) {
                _extraReward.updatePool();
            }

            extraRewards[i].distributeReward(_user, rewardAmount);

        }
    }
```

The major discrepancy here is that, during the reward distribution, if a particular `_extraReward.isSettledIncome() == false`, then the `updatePool()` should be called on this pool. The `distributeAllRewards` has accounted for this piece of logic, but the `harvest` function missed it. Missing this logic will lead to major impact on the fairness of reward distribution to the future users, because some of the specs/data of that pool didn't get updated accordingly.

Recommendation

***: Simply add the `updatePool()` related logic to the `harvest` function, as how it was done in the `distributeAllRewards`.

Client Response

client response : Fixed.

BIO-8:Potential inconsistency when handling reward distribution

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L95-L122
- code/farms/mendiCompoundFarm.sol#L207-L226
- code/farms/mendiCompoundFarm.sol#L276-L295

```
95: /// @notice Remove the reward token from pool
96:     /// @param _rewardTokenAddr The reward token to remove
97:     function removeExtraReward(address _rewardTokenAddr) external onlyOwner {
98:         require(_rewardTokenAddr != address(0), "Invalid reward address");
99:
100:         address[] memory userList = userAddrList.values();
101:
102:         for (uint256 i = 0; i < extraRewards.length; i++) {
103:             if (address(extraRewards[i]) == _rewardTokenAddr) {
104:
105:                 IRewardNew _reward = IRewardNew(_rewardTokenAddr);
106:                 for (uint256 j = 0; j < userList.length; j++) {
107:                     UserInfo storage userInfo = userInfoMap[userList[j]];
108:                     uint256 rewardAmount = _reward.calculateReward(userList[j],
109:                         extraRewards[i].isSettledIncome() ? userInfo.underlyingAmount : 0);
110:
111:                     _reward.distributeReward(userList[j], rewardAmount);
112:                 }
113:
114:                 extraRewards[i] = extraRewards[extraRewards.length - 1];
```

```
115:                 extraRewards.pop();
116:
117:                 break;
118:             }
119:         }
120:
121:         emit RemoveExtraRewardToken(_rewardTokenAddr);
122:     }
```

```
207: /// @notice Calculate the rewards and transfer to user
208:     /// @param _user The user address
209:     function harvest(address _user) external nonReentrant {
210:         require(startTimestamp > 0, "Mining not start!!");
211:         require(_user != address(0), "Farm: invalid user address");
212:
213:         UserInfo storage userInfo = userInfoMap[_user];
214:         require(userInfo.underlyingAmount > 0, "No deposit");
215:
216:         for (uint256 i = 0; i < extraRewards.length; i++) {
217:             IRewardNew _extraReward = extraRewards[i];
218:
219:             uint256 pendingRewards = extraRewards[i].calculateReward(_user,
220:                 _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);
221:
222:             if (pendingRewards > 0) {
223:                 _extraReward.distributeReward(_user, pendingRewards);
224:             }
225:         }
226:     }
```

```
276: /// @notice Distribute all of rewards to user
277:     /// @param _user The user address
278:     function distributeAllRewards(address _user) internal {
279:         UserInfo storage userInfo = userInfoMap[_user];
280:
281:         for (uint256 i = 0; i < extraRewards.length; i++) {
282:             IRewardNew _extraReward = extraRewards[i];
283:
284:             uint256 rewardAmount = extraRewards[i].calculateReward(_user,
285:                 _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);
286:
287:             // Liquidity reward
288:             if (_extraReward.isSettledIncome() == false) {
289:                 _extraReward.updatePool();
290:             }
291:
292:             extraRewards[i].distributeReward(_user, rewardAmount);
293:
294:         }
295:     }
```

Description

***: In `distributeAllRewards`, the rewards are calculated and distributed based on the `isSettledIncome` condition:

1. If it returns `true`, it uses `userInfo.underlyingAmount` (indicating that the user has some stake or underlying asset).
2. If it returns `false`, before calculating the reward, it calls `_extraReward.updatePool()` to presumably refresh state before reward distribution.

```
    uint256 rewardAmount = extraRewards[i].calculateReward(_user,
        _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);

    // Liquidity reward
    if (_extraReward.isSettledIncome() == false) {
        _extraReward.updatePool();
    }

    extraRewards[i].distributeReward(_user, rewardAmount);
```

In `removeExtraReward` and `harvest`, a similar condition is checked, leading to an inconsistency with how rewards are accounted for, especially if a user's status changes before rewards are distributed:

```
//in removeExtraReward function
uint256 rewardAmount = _reward.calculateReward(userList[j],
                        extraRewards[i].isSettledIncome() ? userInfo.underlyingAmount : 0);
_reward.distributeReward(userList[j], rewardAmount);
```

```
// in harvest function
IRewardNew _extraReward = extraRewards[i];

uint256 pendingRewards = extraRewards[i].calculateReward(_user,
    _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);

if (pendingRewards > 0) {
    _extraReward.distributeReward(_user, pendingRewards);
}
```

Recommendation

***: Consider following fix in `removeExtraReward`:

```
function removeExtraReward(address _rewardTokenAddr) external onlyOwner {
        require(_rewardTokenAddr != address(0), "Invalid reward address");

        address[] memory userList = userAddrList.values();

        for (uint256 i = 0; i < extraRewards.length; i++) {
            if (address(extraRewards[i]) == _rewardTokenAddr) {

                IRewardNew _reward = IRewardNew(_rewardTokenAddr);
                for (uint256 j = 0; j < userList.length; j++) {
                    UserInfo storage userInfo = userInfoMap[userList[j]];
                    uint256 rewardAmount = _reward.calculateReward(userList[j],
                        extraRewards[i].isSettledIncome() ? userInfo.underlyingAmount : 0);
                    // Liquidity reward
                    if (extraRewards[i].isSettledIncome() == false) {
                        extraRewards[i].updatePool();
                    }
                    _reward.distributeReward(userList[j], rewardAmount);
                }

                extraRewards[i] = extraRewards[extraRewards.length - 1];
                extraRewards.pop();

                break;
            }
        }

        emit RemoveExtraRewardToken(_rewardTokenAddr);
    }
```

Consider following fix in **`harvest`**:

```
function harvest(address _user) external nonReentrant {
        require(startTimestamp > 0, "Mining not start!!");
        require(_user != address(0), "Farm: invalid user address");

        UserInfo storage userInfo = userInfoMap[_user];
        require(userInfo.underlyingAmount > 0, "No deposit");

        for (uint256 i = 0; i < extraRewards.length; i++) {
            IRewardNew _extraReward = extraRewards[i];

            uint256 pendingRewards = extraRewards[i].calculateReward(_user,
                _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);
            // Liquidity reward
            if (extraRewards[i].isSettledIncome() == false) {
                extraRewards[i].updatePool();
            }
            if (pendingRewards > 0) {
                _extraReward.distributeReward(_user, pendingRewards);
            }
        }
    }
```

Client Response

client response : Fixed.

BIO-9:Potential for Reward Token Drain During Removal

Category	Severity	Client Response	Contributor
Integer Overflow and Underflow	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L97

```
97: function removeExtraReward(address _rewardTokenAddr) external onlyOwner {
```

Description

***: The `removeExtraReward` function attempts to distribute all remaining rewards to users before removing the reward token. However, it doesn't check if there are sufficient reward tokens available in the contract to cover all the distributions. This could lead to a situation where the owner, intentionally or unintentionally, triggers a mass distribution of rewards that the contract can't fulfill.
Edge Case Scenario:
1.The farm contract has accumulated a significant amount of a particular reward token.
2.The owner decides to remove this reward token.
3.When removeExtraReward is called, it attempts to distribute all accrued rewards to all users.
4.If the total accrued rewards exceed the actual balance of reward tokens in the contract, some users may not receive their full rewards, or the function may revert due to insufficient balance.
Impact:
1.Unfair distribution of remaining rewards: Users who are processed earlier in the loop will receive their rewards, while later users might not.
2.Potential for the function to fail mid-execution, leaving the contract in an inconsistent state.
3.If the reward token's distributeReward function doesn't properly handle insufficient balances, it could lead to unexpected behavior or even a complete halt of the removal process.

Recommendation

***: Implement a check to ensure the contract has sufficient balance of the reward token before starting the distribution:

```
function removeExtraReward(address _rewardTokenAddr) external onlyOwner {
    require(_rewardTokenAddr != address(0), "Invalid reward address");

    IERC20 rewardToken = IERC20(_rewardTokenAddr);
    uint256 contractBalance = rewardToken.balanceOf(address(this));

    address[] memory userList = userAddrList.values();
    uint256 totalRewardsToDistribute = 0;

    // First, calculate total rewards to distribute
    for (uint256 i = 0; i < extraRewards.length; i++) {
        if (address(extraRewards[i]) == _rewardTokenAddr) {
            IRewardNew _reward = IRewardNew(_rewardTokenAddr);
            for (uint256 j = 0; j < userList.length; j++) {
                UserInfo storage userInfo = userInfoMap[userList[j]];
                totalRewardsToDistribute += _reward.calculateReward(
                    userList[j], extraRewards[i].isSettledIncome() ? userInfo.underlyingAmount : 0
                );
            }
            break;
        }
    }

    require(contractBalance >= totalRewardsToDistribute, "Insufficient reward balance for distribu
tion");

    // If sufficient balance, proceed with distribution and removal
    // ... (rest of the original function logic)
}
```

2. Consider implementing a cap on the total rewards that can be distributed during removal, or allow for partial distributions if the full amount isn't available.
3. Add proper error handling and event emissions to track any discrepancies between calculated rewards and actual distributions.

This approach ensures that the owner can't inadvertently trigger a distribution that the contract can't fulfill, protecting both the users and the integrity of the reward system.

Client Response

client response : Fixed. Intentional design, for the purpose of gas cost reduction. Instead, we utilize getRemovalRewardAmounts to fetch the reward amount, which allows us to run manual check on the token balance in case an error occurs

BIO-10:Potential DOS risk in the function `distributeAllRewards` and `removeExtraReward`

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L111
- code/farms/mendiCompoundFarm.sol#L278-L295

```
111: _reward.distributeReward(userList[j], rewardAmount);
```

```
278: function distributeAllRewards(address _user) internal {
279:         UserInfo storage userInfo = userInfoMap[_user];
280:
281:         for (uint256 i = 0; i < extraRewards.length; i++) {
282:             IRewardNew _extraReward = extraRewards[i];
283:
284:             uint256 rewardAmount = extraRewards[i].calculateReward(_user,
285:                 _extraReward.isSettledIncome() ? userInfo.underlyingAmount : 0);
286:
287:             // Liquidity reward
288:             if (_extraReward.isSettledIncome() == false) {
289:                 _extraReward.updatePool();
290:             }
291:
292:             extraRewards[i].distributeReward(_user, rewardAmount);
293:
294:         }
295:     }
```

Description

***: In the `distributeAllRewards`, before executing this line `extraRewards[i].distributeReward(_user, rewardAmount);`, the code forgot to check the pre-condition of the `rewardAmount`.
As a comparison, the `harvest` function has the good implementation regarding this. See below:

```
            if (pendingRewards > 0) {
                _extraReward.distributeReward(_user, pendingRewards);
            }
```

But the `distributeAllRewards` forgot to have that check of `if (rewardAmount > 0) {...}`. If the calculated pending reward amount is `0` in any iteration, then `extraRewards[i].distributeReward` should not be called in that iteration. Missing that check can lead to either minor or catastrophic consequences:

- The minor consequence is referring to some extra gas wasted unnecessarily because the `extraRewards[i].distributeReward` execution could have been bypassed in the case of `rewardAmount==0`;
- The catastrophic consequence is referring to that, depending on the specific implementation of the `extraRewards[i].distributeReward` function, if it has a validation like `require(rewardAmount>0, "No reward to distri`

`ute!")`, then the whole `distributeAllRewards` can be DOS'ed for the user. The user is unable to get any rewards from any of the reward pools because the entire loop will revert, not only just about that particular reward pool.

This finding also similarly applies to the relevant reward distribution logic inside the function `removeExtraReward`.

Recommendation

***: Simply add the `if (rewardAmount > 0) {...}` execution flow control to the `distributeAllRewards` function, as how it was done in the `harvest` function.
This recommendation also similarly applies to the relevant reward distribution logic inside the function `removeExtraReward`.

Client Response

client response : Fixed.

BIO-11:Not calling approve(0) before setting a new approval causes the call to revert when used with Tether (USDT)

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L126-L167

```
126: function deposit(uint256 _amount) external payable nonReentrant {
127:         require(startTimestamp > 0, "Farm: mining not start!!");
128:
129:         UserInfo storage userInfo = userInfoMap[msg.sender];
130:
131:         // Distribute rewards to user
132:         distributeAllRewards(msg.sender);
133:
134:         // process WETH
135:         if (address(assetToken) == ethAddr) {
136:             require(_amount == 0, "Deposit invalid token");
137:
138:             if (msg.value > 0) {
139:                 _amount = _amount + msg.value;
140:             }
141:         } else {
142:             require(msg.value == 0, "Deposit invalid token");
143:             if (_amount > 0) {
144:                 TransferHelper.safeTransferFrom(address(assetToken), address(msg.sender), addre
ss(this), _amount);
145:             }
```

```
146:
147:             assetToken.approve(address(mendiCToken), _amount);
148:         }
149:
150:         // Save assets to Mendi
151:         assetToken.approve(address(mendiCToken), _amount);
152:         mendiCToken.mint(_amount);
153:
154:         // Calculate the cToken
155:         uint256 calcCToken = underlyingToCToken(_amount);
156:         userInfo.underlyingAmount += _amount;
157:         userInfo.cTokenAmount += calcCToken;
158:         userInfo.lastDepositTime = block.timestamp;
159:
160:         userAddrList.add(msg.sender);
161:
162:         totalDeposits += _amount;
163:
164:         updateAllRewards(msg.sender, _amount, true);
165:
```

```
166:         emit Deposit(msg.sender, _amount, calcCToken);
167:     }
```

Description

***:** Some tokens do not implement the ERC20 standard properly but are still accepted by most code that accepts ERC20 tokens. For example Tether (USDT)'s **`approve()`** function will revert if the current approval is not zero, to protect against front-running changes of approvals.
In function **`deposit`**, it will call the assetToken's **`approve()`** function for twice:

```
if (address(assetToken) == ethAddr) {
            require(_amount == 0, "Deposit invalid token");

            if (msg.value > 0) {
                _amount = _amount + msg.value;
            }
        } else {
            require(msg.value == 0, "Deposit invalid token");
            if (_amount > 0) {
                TransferHelper.safeTransferFrom(address(assetToken), address(msg.sender), address
(this), _amount);
            }

            assetToken.approve(address(mendiCToken), _amount);
        }

        // Save assets to Mendi
        assetToken.approve(address(mendiCToken), _amount);
```

If the **`assetToken`** is USDT, the second approve will revert.

Recommendation

***:** Use OpenZeppelin's **`SafeERC20`** 's forceApprove instead. Or use method **`safeApprove`** of Lib **`TransferHelper.sol`**

Client Response

client response : Fixed.

BIO-12:Lack of Time-Based Restrictions on Reward Distribution

Category	Severity	Client Response	Contributor
Flash Loan Attacks	Low	Acknowledged	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L219

```
219: uint256 pendingRewards = extraRewards[i].calculateReward(_user,
```

Description

***: The **harvest** function allows users to claim their rewards at any time without any cooldown period or time-based restrictions. This could be exploited in a flash loan attack where an attacker deposits a large amount, immediately harvests rewards, and then withdraws, all within a single transaction.

Proof of Concept:

1.Attacker takes a flash loan for a large amount of the asset token.
2.Calls `deposit` to add a significant amount to the farm.
3.Immediately calls `harvest` to claim rewards based on this large deposit.
4.Withdraws the deposited amount.
5.Repays the flash loan.
The attacker could potentially extract a disproportionate amount of rewards in a single transaction, disrupting the intended token economics of the farm.
Impact:
This vulnerability could lead to:
1.Unfair distribution of rewards
2.Draining of the reward pool
3.Devaluation of the reward token
4.Loss of faith in the protocol by legitimate users.

Recommendation

***: Implement a time-based restriction on reward harvesting. For example:

```
mapping(address => uint256) public lastHarvestTime;
uint256 public constant HARVEST_COOLDOWN = 1 days;

function harvest(address _user) external nonReentrant {
    require(startTimestamp > 0, "Mining not start!!");
    require(_user != address(0), "Farm: invalid user address");
    require(block.timestamp >= lastHarvestTime[_user] + HARVEST_COOLDOWN, "Harvest cooldown not me
t");

    UserInfo storage userInfo = userInfoMap[_user];
    require(userInfo.underlyingAmount > 0, "No deposit");

    for (uint256 i = 0; i < extraRewards.length; i++) {
        IRewardNew _extraReward = extraRewards[i];

        uint256 pendingRewards =
            extraRewards[i].calculateReward(_user, _extraReward.isSettledIncome() ? userInfo.under
lyingAmount : 0);

        if (pendingRewards > 0) {
            _extraReward.distributeReward(_user, pendingRewards);
        }
    }

    lastHarvestTime[_user] = block.timestamp;
}
```

This change introduces a cooldown period between harvests, making it impossible to perform flash loan attacks that rely on immediate reward claiming. It significantly increases the security of the reward distribution mechanism while still allowing legitimate users to claim their rewards regularly.

Client Response

client response : Acknowledged. This is intentional design for the purpose of security enhancement.

BIO-13:In the event of default happening on the Mendi's side, the farming contract will allow the fast runners escape with no loss and slow hands bear all the loss

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L181-L200

```
181: mendiCToken.redeemUnderlying(_amount);
182:
183:         uint256 reduceCTokenAmount = underlyingToCToken(_amount);
184:
185:         if (userInfo.cTokenAmount > reduceCTokenAmount) {
186:             userInfo.cTokenAmount -= reduceCTokenAmount;
187:         } else {
188:             userInfo.cTokenAmount = 0;
189:         }
190:
191:         userInfo.underlyingAmount -= _amount;
192:         userInfo.lastDepositTime = block.timestamp;
193:
194:         totalDeposits -= _amount;
195:
196:         if (address(assetToken) == ethAddr) {
197:             TransferHelper.safeTransferETH(msg.sender, _amount);
198:         } else {
199:             TransferHelper.safeTransfer(address(assetToken), msg.sender, _amount);
200:         }
```

Description

***: The issue is in the situation when the Mendi's side is having defaults, the **`cToken`** to **`assetToken`** exchange rate is negatively affected. For example, Alice initially deposited **`100 USDC`** and Mendi minted **`100 mendiCUSDC`**, so that Alice's **`userInfo.underlyingAmount`** and **`userInfo.cTokenAmount`** both should be 100 at the moment (decimals ignored).
Then, some defaults have happened and the Mendi's lending protocol lost a significant amount of **`USDC`**, in this case, the **`exchangeRate`** is negatively affected. Now, to redeem for **`100 USDC`**, the **`100 mendiCUSDC`** is not enough. Let's assume now it needs **`150 mendiCUSDC`** to redeem for **`100 USDC`**.
Now, Alice is the first one to hear about this bad news and she immediately reacts to it. She calls **`withdraw`** for all her **`100 USDC`** from the farming contract. Look at these lines:

```
        mendiCToken.redeemUnderlying(_amount);

        uint256 reduceCTokenAmount = underlyingToCToken(_amount);

        if (userInfo.cTokenAmount > reduceCTokenAmount) {
            userInfo.cTokenAmount -= reduceCTokenAmount;
        } else {
            userInfo.cTokenAmount = 0;
        }

        userInfo.underlyingAmount -= _amount;
        userInfo.lastDepositTime = block.timestamp;

        totalDeposits -= _amount;

        if (address(assetToken) == ethAddr) {
            TransferHelper.safeTransferETH(msg.sender, _amount);
        } else {
            TransferHelper.safeTransfer(address(assetToken), msg.sender, _amount);
        }
```

First, the farming contract can successfully redeem for `100 USDC` out of Mendi, because the farming contract holds many users' `mendiCUSDC` so it got enough amount to burn for Alice's withdraw. At this step, `150 mendiCUSDC` got burned from the farming contract, which is also the `reduceCTokenAmount`.
However, note that Alice's `userInfo.cTokenAmount` is only `100`, smaller than `150`, so the code will make it `userInfo.cTokenAmount = 0`. But Alice doesn't care, because in the next step `TransferHelper.safeTransfer(address(assetToken), msg.sender, _amount)` will send all the `100 USDC` to her. Alice gets to walk away with her 100 bucks, no loss to her principal, but the farming contract just lost the `50` extra `mendiCUSDC`. As this continues, more fast runner users quit with no loss, eventually the last cohort of users who react slowly will find themselve not able to withdraw the full amount of their principals at all, because the farming contract no longer holds enough cTokens.

Recommendation

***: When there is bad debt happening, the loss should be socialized accordingly no matter who reacts fast and who is slow.
Recommend to re-write the `withdraw` function and do not use the `_amount` of the `underlying asset` as the input; instead, use the `_amount` of the `cToken` as the input. Then, interact with Mendi's `mendiCToken.redeem` function instead of the `mendiCToken.redeemUnderlying` function.

Client Response

client response : Acknowledged. This finding indicated an extreme incidence that rarely happens, therefore insignificant.

BIO-14:Arbitrary Timestamp Setting

Category	Severity	Client Response	Contributor
Privilege Related	Low	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L236
- code/farms/mendiCompoundFarm.sol#L236-L241

```
236: function setStartTimestamp(uint256 _timestamp) external onlyOwner {
```

```
236: function setStartTimestamp(uint256 _timestamp) external onlyOwner {
237:         require(startTimestamp == 0, "Farm: already started");
238:
239:         startTimestamp = _timestamp;
240:         emit EventSetStartTimestamp(_timestamp);
241:     }
```

Description

***: The function `setStartTimestamp` allows the owner to set the start timestamp to any value, including a time in the past. This could be exploited in several ways:

1. Backdating: The owner could set the start time to a point in the past, potentially manipulating reward calculations or enabling premature reward distributions.
2. Future Dating: Setting a future timestamp could delay the start of farming operations unexpectedly.
3. Inconsistency with `startMining()`: This function overlaps in functionality with `startMining()` but behaves differently, which could lead to confusion or inconsistent contract state.

***: In the farming contract there are two ways to start the farming: the `startMining` which starts the farming at the current `block.timestamp`, and the `setStartTimestamp` which is meant to pre-setup a starting time to be in the future. However, the `setStartTimestamp` forgot to add an validation to ensure the input `_timestamp` is sometime in the future but NOT sometime in the past. If that wrong timestamp input gets accepted, it can affect the reward token's minting and distributing which is unfair to the users participating in the farming; it will also impact all other parts of the protocol which may use the public `startTimestamp` as a dependency for whatever calculations.
In addition, once a wrong input `_timestamp` in the past was taken, there is no way to correct it anymore, because the check of `require(startTimestamp == 0, "Farm: already started");` shall always revert any more calls on the `setStartTimestamp`. Thus, it's super crucial to have an validation on the `_timestamp` to ensure it's not a timestamp in the past.

Recommendation

***: Rewrite the function to enforce stricter rules:

```
/// @notice Set the farm start timestamp
/// @param _timestamp The farm start timestamp(seconds)
function setStartTimestamp(uint256 _timestamp) external onlyOwner {
    require(startTimestamp == 0, "Farm: already started");
    require(_timestamp >= block.timestamp, "Cannot set start time in the past");
    require(_timestamp <= block.timestamp + 30 days, "Start time too far in the future");
    startTimestamp = _timestamp;
    emit EventSetStartTimestamp(_timestamp);
}
```

***: To mitigate this vulnerability, it's crucial to add a validation step in the `setStartTimestamp` function to ensure that the `_timestamp` is in the future compared to the current blockchain's timestamp. This can be done by adding a line of code like:

```
require(_timestamp > block.timestamp, "Farm: start timestamp must be in the future");
```

This check will prevent setting a past timestamp, thereby ensuring the start-time-related functionalities execute as intended at the correct future time.

Client Response

client response : Fixed.

BIO-15: `startMining` Function Lacks Event Emission

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L229-L232

```
229: function startMining() public onlyOwner {
230:         require(startTimestamp == 0, "Farm: mining already started");
231:         startTimestamp = block.timestamp;
232:     }
```

Description

***: The `startMining` function in the contract, which initiates the mining process, does not emit any event:

```
    function startMining() public onlyOwner {
        require(startTimestamp == 0, "Farm: mining already started");
        startTimestamp = block.timestamp;
    }
```

Recommendation

***: Add an event to the `startMining` function

Client Response

client response : Fixed.

BIO-16:Unused Libs

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L7

```
7: import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

Description

***: In the contract `BagfulMendiCompoundFarm`, there has imported the `SafeERC20` lib:

```
contract BagfulMendiCompoundFarm is Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable,
IFarm {
    using SafeERC20 for IERC20;
    // ...snip...
}
```

However, the functions in `SafeERC20` lib have never been used, just used the normal `ERC20` functions like `balanceOf` and `approve`.
And token transfer function is implemented through `TransferHelper.sol`.
So, it is an unnecessary lib which enlarges the size of bytcode costing more gas.

Recommendation

***: Consider removing the `SafeERC20` lib for gas saving.

```
- import "../comm/TransferHelper.sol";

contract BagfulMendiCompoundFarm is Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable,
IFarm {
-   using SafeERC20 for IERC20;
    // ...snip...
}
```

Client Response

client response : Fixed.

BIO-17:The initialize function lacks zero-address checks

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L65-L78

```
65: function initialize(
66:         address _assets,
67:         address _mendiCToken,
68:         address _ethAddr
69:     ) public initializer {
70:         __Ownable_init(msg.sender);
71:         __ReentrancyGuard_init();
72:
73:         assetToken = IERC20(_assets);
74:         mendiCToken = IMendiCToken(_mendiCToken);
75:         ethAddr = _ethAddr;
76:
77:         totalDeposits = 0;
78:     }
```

Description

***: Since the functions addExtraReward, removeExtraReward, and setMendiCToken in the contract can only be called by the owner and all have zero-address checks for the corresponding parameters, the lack of zero-address checks for the parameters _assets, _mendiCToken, and _ethAddr in the initialize function is a security issue.
If there are problems in these three parameters, it will affect the normal operation of the functions deposit and withdraw.

Recommendation

***: Add zero-address checks.

Client Response

client response : Fixed.

BIO-18:Replace `abi.encodeWithSelector` with `abi.encodeCall` which keeps the code typo/type safe

Category	Severity	Client Response	Contributor
Language Specific	Informational	Fixed	***

Code Reference

- code/comm/TransferHelper.sol#L6
- code/comm/TransferHelper.sol#L11
- code/comm/TransferHelper.sol#L16

```
6: (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x095ea7b3, to, value));
```

```
11: (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb, to, value));
```

```
16: (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd, from, to, valu
e));
```

Description

***: When using `abi.encodeWithSelector`, it is possible to provide parameters that are not of the correct type for the function.
To avoid these pitfalls, it would be best to use abi.encodeCall instead.

Recommendation

***: Consider following fix:

```
(bool success, bytes memory data) = token.call(abi.encodeCall(0x095ea7b3, to, value));
```

```
(bool success, bytes memory data) = token.call(abi.encodeCall(0xa9059cbb, to, value));
```

```
(bool success, bytes memory data) = token.call(abi.encodeCall(0x23b872dd, from, to, value));
```

Client Response

client response : Fixed.

BIO-19:Redundant approve() Call in the deposit() Function

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L126-L151

```
126: function deposit(uint256 _amount) external payable nonReentrant {
127:         require(startTimestamp > 0, "Farm: mining not start!!");
128:
129:         UserInfo storage userInfo = userInfoMap[msg.sender];
130:
131:         // Distribute rewards to user
132:         distributeAllRewards(msg.sender);
133:
134:         // process WETH
135:         if (address(assetToken) == ethAddr) {
136:             require(_amount == 0, "Deposit invalid token");
137:
138:             if (msg.value > 0) {
139:                 _amount = _amount + msg.value;
140:             }
141:         } else {
142:             require(msg.value == 0, "Deposit invalid token");
143:             if (_amount > 0) {
144:                 TransferHelper.safeTransferFrom(address(assetToken), address(msg.sender), addre
ss(this), _amount);
145:             }
```

```
146:
147:             assetToken.approve(address(mendiCToken), _amount);
148:         }
149:
150:         // Save assets to Mendi
151:         assetToken.approve(address(mendiCToken), _amount);
```

Description

***: The `deposit` function calls the `approve` function for the same token with the same amount successively when the `address(assetToken)` is not the `ethAddr`:

```
function deposit(uint256 _amount) external payable nonReentrant {
    ...
    // process WETH
    if (address(assetToken) == ethAddr) {
        ...
    } else {
        ...
        assetToken.approve(address(mendiCToken), _amount);
    }

    // Save assets to Mendi
    assetToken.approve(address(mendiCToken), _amount);
```

The same approve action in the `else` branch turns redundant and useless.

Recommendation

***: Consider removing the redundant and useless approve action in the `else` branch.

Client Response

client response : Fixed.

BIO-20:Ownership change should use two-step process

Category	Severity	Client Response	Contributor
Privilege Related	Informational	Acknowledged	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L4

```
4: import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
```

Description

***: The contracts `mendiCompoundFarm.sol` does not implement a 2-Step-Process for transferring ownership.
So ownership of the contract can easily be lost when making a mistake when transferring ownership.
Since the privileged roles have critical function roles assigned to them. Assigning the ownership to a wrong user can be disastrous.
So Consider using the `Ownable2StepUpgradeable` contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/Ownable2StepUpgradeable.sol) instead.
The way it works is there is a `transferOwnership` to transfer the ownership and `acceptOwnership` to accept the ownership. Refer the above Ownable2StepUpgradeable.sol for more details.

Recommendation

***: Implement 2-Step-Process for transferring ownership via `Ownable2StepUpgradeable`.

Client Response

client response : Acknowledged. We will implement the two-step process in the future.

BIO-21:Missing __gap variable in the contract BagfulMendiCompoundFarm

Category	Severity	Client Response	Contributor
DOS	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L14

```
14: contract BagfulMendiCompoundFarm is Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeabl
e, IFarm {
```

Description

***: In upgradeable contracts, maintaining a consistent storage layout across versions is crucial to avoid storage collisions, which can lead to data loss or corruption. When using proxy patterns (e.g., OpenZeppelin's upgradeable contracts), new state variables in future versions must align with the existing storage layout. Without proper storage reservation, new variables can overwrite existing slots, causing issues like lost user balances or corrupted mappings. To prevent this, OpenZeppelin's Initializable module suggests including a __gap array. This reserved storage acts as a buffer, ensuring that future upgrades don't unintentionally overwrite existing data, especially in contracts that inherit from multiple base contracts.

Recommendation

***: It is strongly recommended to add a __gap variable to this contract. This will prevent future storage collisions during upgrades and ensure the contract remains upgradeable without risk of storage corruption. The following line should be added towards the end of the contract to reserve these storage slots:

```
uint256[50] private __gap;
```

Client Response

client response : Fixed.

BIO-22:Lack of Pause Functionality

Category	Severity	Client Response	Contributor
Privilege Related	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L72

```
72:
```

Description

***: The **BagfulMendiCompoundFarm** contract lacks a pause functionality, which limits the ability of the contract owner to temporarily suspend operations in the case of emergencies, bugs, or security vulnerabilities. Without this functionality, if an exploit or unexpected behavior is discovered, it could continue to affect the contract and its users until a permanent fix is implemented. This could lead to a significant loss of funds or incorrect reward distributions.

"Affects critical functions like deposit(), withdraw(), harvest(), and potentially other user-facing operations."

Recommendation

***: Add pause functionality using OpenZeppelin's **PausableUpgradeable** contract to allow the contract owner to suspend critical operations (such as deposit(), withdraw(), and harvest()) during emergencies. This ensures that interactions can be temporarily halted to prevent further damage while a fix or upgrade is deployed.

1. Update initialize :

Reference (https://forum.openzeppelin.com/t/is-it-mandatory-to-call-pausable-init-or-any-init-functions-within-upgradable-contracts-what-happens-if-you-dont/40565)

```
  function initialize(address _assets, address _mendiCToken, address _ethAddr) public initializer
{
      __Ownable_init();
      __ReentrancyGuard_init();
      __Pausable_init();

      assetToken = IERC20(_assets);
      mendiCToken = IMendiCToken(_mendiCToken);
      ethAddr = _ethAddr;

      totalDeposits = 0;
  }
```

2. Add **whenNotPaused** Modifier to Critical Functions: Apply the whenNotPaused modifier to functions where interactions with the contract can be halted during emergencies:

```
function deposit(uint256 _amount) external payable nonReentrant whenNotPaused {
    // Existing logic for deposit
}

function withdraw(uint256 _amount) external nonReentrant whenNotPaused {
    // Existing logic for withdraw
}

function harvest(address _user) external nonReentrant whenNotPaused {
    // Existing logic for harvesting rewards
}

function startMining() public onlyOwner whenNotPaused {
    require(startTimestamp == 0, "Farm: mining already started");
    startTimestamp = block.timestamp;
}
```

3. Add **pause()** and **unpause()** Functions: Define functions that allow the contract owner to pause and unpause the contract:

```
function pause() external onlyOwner {
    _pause();
}

function unpause() external onlyOwner {
    _unpause();
}
```

Client Response

client response : Fixed.

BIO-23:Initializing totalDeposits to 0 is unnecessary in the context of the `initialize` function

Category	Severity	Client Response	Contributor
Code Style	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L64-L78

```
64: /// @notice Initialize the farm
65:     function initialize(
66:         address _assets,
67:         address _mendiCToken,
68:         address _ethAddr
69:     ) public initializer {
70:         __Ownable_init(msg.sender);
71:         __ReentrancyGuard_init();
72:
73:         assetToken = IERC20(_assets);
74:         mendiCToken = IMendiCToken(_mendiCToken);
75:         ethAddr = _ethAddr;
76:
77:         totalDeposits = 0;
78:     }
```

Description

***: In Solidity, state variables that are not explicitly initialized are automatically set to their default values. For uint types like `totalDeposits`, the default value is already 0. Therefore, assigning `totalDeposits = 0;` is redundant and adds unnecessary code.

Recommendation

***: Consider removing the code `totalDeposits = 0;` in the `initialize` function.

Client Response

client response : Fixed.

BIO-24:Incompatibility With Deflationary Tokens

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L126-L167

```
126: function deposit(uint256 _amount) external payable nonReentrant {
127:         require(startTimestamp > 0, "Farm: mining not start!!");
128:
129:         UserInfo storage userInfo = userInfoMap[msg.sender];
130:
131:         // Distribute rewards to user
132:         distributeAllRewards(msg.sender);
133:
134:         // process WETH
135:         if (address(assetToken) == ethAddr) {
136:             require(_amount == 0, "Deposit invalid token");
137:
138:             if (msg.value > 0) {
139:                 _amount = _amount + msg.value;
140:             }
141:         } else {
142:             require(msg.value == 0, "Deposit invalid token");
143:             if (_amount > 0) {
144:                 TransferHelper.safeTransferFrom(address(assetToken), address(msg.sender), addre
ss(this), _amount);
145:             }
```

```
146:
147:             assetToken.approve(address(mendiCToken), _amount);
148:         }
149:
150:         // Save assets to Mendi
151:         assetToken.approve(address(mendiCToken), _amount);
152:         mendiCToken.mint(_amount);
153:
154:         // Calculate the cToken
155:         uint256 calcCToken = underlyingToCToken(_amount);
156:         userInfo.underlyingAmount += _amount;
157:         userInfo.cTokenAmount += calcCToken;
158:         userInfo.lastDepositTime = block.timestamp;
159:
160:         userAddrList.add(msg.sender);
161:
162:         totalDeposits += _amount;
163:
164:         updateAllRewards(msg.sender, _amount, true);
165:
```

```
166:         emit Deposit(msg.sender, _amount, calcCToken);
167:     }
```

Description

***:** In `mendiCompoundFarm`, the function `deposit` will call `safeTransferFrom` to transfer token to the current contract and record the token balance:

```
if (_amount > 0) {
                TransferHelper.safeTransferFrom(address(assetToken), address(msg.sender), address
(this), _amount);
            }
```

```
totalDeposits += _amount;
```

If the `assetToken` is a deflationary token, the input param `_amount` may not be equal to the received amount due to the charged transaction fee. That means the `totalDeposits` will be wrong. When calling `withdraw` function, it may fail due to insufficient tokens.

Recommendation

***:** Consider following fix:

```
uint256 before = IERC20(assetToken).balanceOf(address(this));
TransferHelper.safeTransferFrom(address(assetToken), address(msg.sender), address(this), _amount);
_amount = IERC20(assetToken).balanceOf(address(this)) - before;
```

Client Response

client response : Fixed.

BIO-25:An Array's length Should Be Cached To Save Gas in Loops

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	***

Code Reference

- code/farms/mendiCompoundFarm.sol#L85

```
85: for (uint256 i = 0; i < extraRewards.length; i++) {
```

Description

***: The contract `BagfulMendiCompoundFarm` is using the state variable `extraRewards` multiple times in the loops in the function `addExtraReward` , `removeExtraReward` ,etc.
`SLOADs` are expensive (100 gas after the 1st one) compared to `MLOAD` / `MSTORE` (3 gas each).

Recommendation

***: It is recommended to store the array's length in a variable before the for-loop.
For instance:

```
function addExtraReward(address _rewardTokenAddr) external onlyOwner {
        require(_rewardTokenAddr != address(0), "Invalid reward address");
        uint256 len = extraRewards.length;
        for (uint256 i = 0; i < len; i++) {
        ..
    }
}
```

Client Response

client response : Fixed.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.