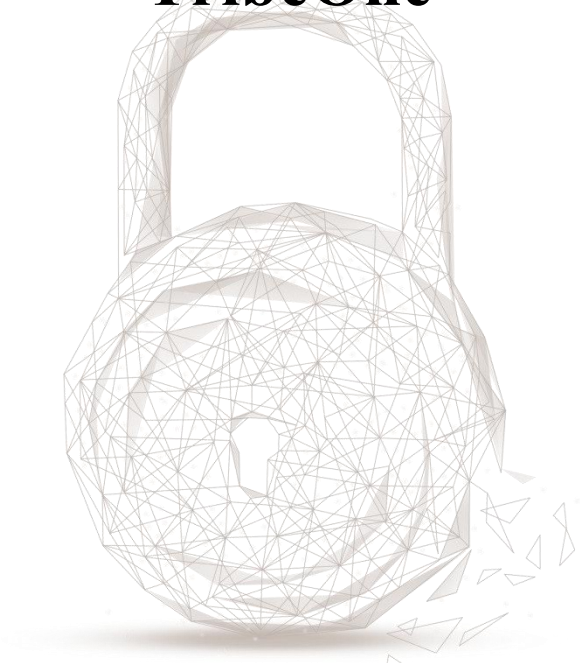
BEOSIN
Blockchain Security

Smart Contract Audit Report

for

TribeOne



Audit Number: 202201171430

Contract Name: TribeOne

Deployment Platform: Ethereum

Github Link: https://github.com/TribeOneDefi/TribeOne-NFT-Loan-Contract

Commit Hash:

16a6a8619c67f1d1bcbc81e18f6224d4cf8a01aa (Initial)

cbc1190ef6fcea7de4466f0b2b3bd7391912efee **(**Final**)**

Audit Start Date: 2021.12.23

Audit Completion Date: 2022.01.17

Audit Team: Beosin Technology Co. Ltd.

Audit Results Overview

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of TribeOne project, including Coding Conventions, General Vulnerability and Business Security. **After auditing, the TribeOne project was found to have 31 risk items: 1 High-risk, 12 Medium-risks, 9 Low-risks and 9 Info. As of the completion of the audit, all risk items have been fixed or properly handled.** The following is the detailed audit information for this project.

Index	Risk items	Risk level	Fix results status
AM-1	Incorrect variable assignment	High	Fixed
AM-2	Centralisation risk	Medium	Acknowledged
AM-3	Owner has a high authority	Medium	Acknowledged
AM-4	Lack of permission checks in *collectInstallment* function	Low	Fixed
AM-5	The keyword emit is not used	Info	Fixed
AM-6	Trigger event recommended when updating twap address	Info	Fixed
MSW-1	Centralisation risk	Low	Acknowledged
MSW-2	Event trigger parameter error	Info	Fixed
MSW-3	Lack of zero address checks	Info	Fixed
MSW-4	Public function that could be declared as external	Info	Fixed
TBO-1	Repeatedly counting the number of late payments	Medium	Fixed
TBO-2	Uninitialised variables	Medium	Fixed
TBO-3	*expectedLastPaymentTime* function implementation flaws	Medium	Fixed
TBO-4	Repeatedly sending tokens	Medium	Fixed
TBO-5	Incorrect number of tokens returned	Medium	Fixed
TBO-6	Incorrect time judgement	Medium	Fixed
TBO-7	Incorrect calculation of late penalty fee	Medium	Fixed
TBO-8	Incorrect fee receiving address	Medium	Fixed
TBO-9	Incorrect token transfer	Medium	Fixed
TBO-10	Incorrect penalty fee calculation	Medium	Fixed
TBO-11	Variable modification without cap	Low	Fixed
TBO-12	*createLoan* function implementation flaws	Low	Fixed
TBO-13	Incorrect highest price determination	Low	Fixed
TBO-14	Lack of permission checks	Low	Fixed
TBO-15	Lack of collateral token value determination in *createLoan*	Low	Acknowledged

	function		
TBO-16	Failure to consider critical conditions when refusing a loan	Low	Fixed
TBO-17	Wrong modifier used	Low	Fixed
TBO-18	Trigger the wrong event	Info	Fixed
TBO-19	Lack of length judgement for input parameters	Info	Fixed
TBO-20	Test code not modified	Info	Fixed
TBO-21	Lack of zero address checks	Info	Fixed

Table1.Key Audit Findings

Risk description:

- Item AM-2 is not fixed and may cause the contract owner to be able to modify the _consumer address, twapOracles mapping and other configuration parameters in the AssetManager contract without restriction.
- Item AM-3 is not fixed and may cause the contract owner to withdraw any tokens in the contract (as of now all the funds to enable the loan functionality is funded from the TribeOne fundraise treasury. That means all the funds in the assetManager contract are not community funds but are the protocol funds, thus the project party has retained the *withdrawAsset* functionality.).
- Item MSW-1 is not fixed and may cause the ETH sent to the TribeOne contract does not match the actual amount refunded by the _agent address.
- Item TBO-15 is not fixed and may cause users to enter malicious data for loan applications. The project party are checking if the loan is valid in the back-end side first and then only approving it.

Findings

[AM-1 High] Incorrect variable assignment

Description: The result returned by calling the *consult* function in the *isValidAutomaticLoan* function of the AssetManager contract contains 18 decimal places, and the automaticLoanLimit is set to 200.

```
22      mapping(address => bool) private availableLoanAsset;
23      mapping(address => bool) private availableCollateralAsset;
24      address private _consumer;
25      uint256 public automaticLoanLimit = 200; // For now we allows NFTs only below 200 usd price
```

Figure 1 source code of AssetManager contract

Fix recommendations: It is recommended to change the value of the variable automaticLoanLimit to "200*10**IERC20(USDC).decimals()".

Fix results: Fixed. It has been fixed at the time of use.

```
114     function isValidAutomaticLoan(address _asset, uint256 _amountIn) external view override returns (bool) {
115         require(availableLoanAsset[_asset], "AssetManager: Invalid loan asset");
116         uint256 usdcAmount;
117         if (_asset == USDC) {
118             usdcAmount = _amountIn;
119         } else {
120             address _twap = twapOracles[_asset];
121             require(_twap != address(0), "AssetManager: Twap oracle was not set");
122 
123             if (_asset == address(0)) {
124                 _asset = WETH;
125             }
126             usdcAmount = ITwapOraclePriceFeed(_twap).consult(_asset, _amountIn);
127         }
128 
129         return usdcAmount <= automaticLoanLimit * (10**IERC20Metadata(USDC).decimals());
130     }
```

Figure 2 source code of *isValidAutomaticLoan* function

[AM-2 Medium] Centralisation risk

Description: In the AssetManager contract, the owner can call *setConsumer, removeAvailableLoanAsset*, *removeAvailableCollateralAsset, setLoanAssetTwapOracle , setAutomaticLoanLimit*, *addAvailableLoanAsset,* and *addAvailableCollateralAsset* funtcions.

```
function addAvailableLoanAsset(address _asset) external onlyOwner nonReentrant {
    require(!availableLoanAsset[_asset], "Already available");
    availableLoanAsset[_asset] = true;
    emit AddAvailableLoanAsset(msg.sender, _asset);
}

function removeAvailableLoanAsset(address _asset) external onlyOwner nonReentrant {
    require(availableLoanAsset[_asset], "Already removed");
    availableLoanAsset[_asset] = false;
    emit RemoveAvailableLoanAsset(msg.sender, _asset);
}

function addAvailableCollateralAsset(address _asset) external onlyOwner nonReentrant {
    require(!availableCollateralAsset[_asset], "Already available");
    availableCollateralAsset[_asset] = true;
    emit AddAvailableCollateralAsset(msg.sender, _asset);
}

function removeAvailableCollateralAsset(address _asset) external onlyOwner nonReentrant {
    require(availableCollateralAsset[_asset], "Already removed");
    availableCollateralAsset[_asset] = false;
    emit RemoveAvailableCollateralAsset(msg.sender, _asset);
}
```

Figure 3 source code of related functions

Fix recommendations: It is recommended to use multi-signature wallet, DAO or TimeLock contract as the contract owner.

Fix results: Acknowledged. For functions like listing loan, relay loan, and dispersing funds from the contract to buy an NFT the project party has implemented multi-sig owner wallets - reducing the potential of exploitation. However they are also developing a DAO contract for handling all the admin activities mentioned above like *requestETH*, etc. Once its setup all ownership will be shifted accordingly. Till then they have kept the functionalities with multi-sig owner wallet access while keeping the admin key very safe behind very secure kubectl environments with database vaults.

[AM-3 Medium] Owner has a high authority

Description: In the AssetManager contract, the *withdrawAsset* function can extract all the tokens in the contract. The owner can also modify the _consumer address at will, and the _consumer can call *requestETH* and *requestToken* functions to extract any tokens in the contract. There is a problem with excessive owner permissions.

```
83      function setConsumer(address _consumer_) external onlyOwner {
84          require(_consumer_ != _consumer, "Already set as consumer");
85          require(_consumer_ != address(0), "ZERO_ADDRESS");
86          _consumer = _consumer_;
87
88          emit SetConsumer(msg.sender, _consumer_);
89      }
```

Figure 4 source code of *setConsumer* function

```
118     function requestETH(address _to, uint256 _amount) external override onlyConsumer {
119         require(address(this).balance >= _amount, "Asset Manager: Insufficient balance");
120         TribeOneHelper.safeTransferETH(_to, _amount);
121         emit TransferAsset(msg.sender, _to, address(0), _amount);
122     }
123
124     function requestToken(
125         address _to,
126         address _token,
127         uint256 _amount
128     ) external override onlyConsumer {
129         require(IERC20(_token).balanceOf(address(this)) >= _amount, "Asset Manager: Insufficient balance");
130         TribeOneHelper.safeTransfer(_token, _to, _amount);
131         emit TransferAsset(msg.sender, _to, _token, _amount);
132     }
```

Figure 5 source code of *requestETH* and *requestToken* functions

```
134     function withdrawAsset(
135         address _to,
136         address _token,
137         uint256 _amount
138     ) external onlyOwner {
139         require(_to != address(0), "ZERO Address");
140         if (_token == address(0)) {
141             _amount = address(this).balance;
142             TribeOneHelper.safeTransferETH(msg.sender, _amount);
143         } else {
144             TribeOneHelper.safeTransfer(_token, msg.sender, _amount);
145         }
146
147         WithdrawAsset(_to, _token, _amount);
148     }
```

Figure 6 source code of *withdrawAsset* function

Fix recommendations: It is recommended to use multi-signature wallets, DAO or TimeLock contracts as contract owners.

Fix results: Acknowledged. As of now all the funds to enable the loan functionality is funded from the TribeOne fundraise treasury. That means all the funds in the assetManager contract are not community funds but are the protocol funds, thus the project party have retained the *withdrawAsset* functionality.

[AM-4 Low] Lack of permission checks in collectInstallment function

Description: The *collectInstallment* function does not check the permissions of the caller.

```
165     function collectInstallment(
166         address _currency,
167         uint256 _amount,
168         uint256 _interest,
169         bool _collateral
170     ) external payable override {
171         if (_currency == address(0)) {
172             require(msg.value == _amount, "Wrong msg.value");
173         } else {
174             TribeOneHelper.safeTransferFrom(_currency, msg.sender, address(this), _amount);
175         }
176         // We will supplement more detail in V2
177         // 80% interest will go to Funding pool rewarder contract, 20% wil be burn
178     }
```

Figure 7 source code of *collectinstallment* function (Unfixed)

Fix recommendations: It is recommended that this be amended to be invoked only by the TribeOne contract to avoid unintended consequences.

Fix results: Fixed.

```
164     function collectInstallment(
165         address _currency,
166         uint256 _amount,
167         uint256 _interest,
168         bool _collateral
169     ) external payable override onlyConsumer {
170         if (_currency == address(0)) {
171             require(msg.value == _amount, "Wrong msg.value");
172         } else {
173             TribeOneHelper.safeTransferFrom(_currency, msg.sender, address(this), _amount);
174         }
175         // We will supplement more detail in V2
176         // 80% interest will go to Funding pool rewarder contract, 20% wil be burn
177         // If _collateral is true, then we transfer whole amount to funding pool
178     }
```

Figure 8 source code of *collectinstallment* function (Fixed)

[AM-5 Info] The keyword emit is not used

Description: The *withdrawAsset* function in the AssetManager contract uses a non-standard form of event triggering code internally.

```
134     function withdrawAsset(
135         address _to,
136         address _token,
137         uint256 _amount
138     ) external onlyOwner {
139         require(_to != address(0), "ZERO Address");
140         if (_token == address(0)) {
141             _amount = address(this).balance;
142             TribeOneHelper.safeTransferETH(msg.sender, _amount);
143         } else {
144             TribeOneHelper.safeTransfer(_token, msg.sender, _amount);
145         }
146 
147         WithdrawAsset(_to, _token, _amount);
148     }
```

Figure 9 source code of *withdrawAsset* function (Unfixed)

Fix recommendations: It is recommended to use the emit keyword for event triggering.

Fix results: Fixed.

```
148     function withdrawAsset(
149         address _to,
150         address _token,
151         uint256 _amount
152     ) external onlyOwner {
153         require(_to != address(0), "ZERO Address");
154         if (_token == address(0)) {
155             _amount = address(this).balance;
156             TribeOneHelper.safeTransferETH(msg.sender, _amount);
157         } else {
158             TribeOneHelper.safeTransfer(_token, msg.sender, _amount);
159         }
160 
161         emit WithdrawAsset(_to, _token, _amount);
162     }
```

Figure 10 source code of *withdrawAsset* function (Fixed)

[AM-6 Info] Trigger event recommended when updating twap address

Description: The event was not triggered when the twap address was updated.

```
92    function setLoanAssetTwapOracle(address _asset, address _twap) external onlyOwner nonReentrant {
93        require(availableLoanAsset[_asset], "AssetManager: Invalid loan asset");
94        address token0 = ITwapOraclePriceFeed(_twap).token0();
95        address token1 = ITwapOraclePriceFeed(_twap).token1();
96        if (_asset == address(0)) {
97            require((token0 == WETH && token1 == USDC) || (token0 == USDC && token1 == WETH), "AssetManager: Invalid twap");
98        } else {
99            require((token0 == _asset && token1 == USDC) || (token0 == USDC && token1 == _asset), "AssetManager: Invalid twap");
100       }
101
102       twapOracles[_asset] = _twap;
103   }
```

Figure 11 source code of *approveLoan* function (Unfixed)

Fix recommendations: It is recommended to trigger the twap address update event.

Fix results: Fixed.

```
93    function setLoanAssetTwapOracle(address _asset, address _twap) external onlyOwner nonReentrant {
94        require(availableLoanAsset[_asset], "AssetManager: Invalid loan asset");
95        address token0 = ITwapOraclePriceFeed(_twap).token0();
96        address token1 = ITwapOraclePriceFeed(_twap).token1();
97        if (_asset == address(0)) {
98            require((token0 == WETH && token1 == USDC) || (token0 == USDC && token1 == WETH), "AssetManager: Invalid twap");
99        } else {
100           require((token0 == _asset && token1 == USDC) || (token0 == USDC && token1 == _asset), "AssetManager: Invalid twap");
101       }
102
103       twapOracles[_asset] = _twap;
104       emit SetTwapOracle(_asset, _twap, msg.sender);
105   }
```

Figure 12 source code of *approveLoan* function (Fixed)

[MSW-1 Low] Centralisation risk

Description: If the token paid for the purchase of an NFT is ETH, the ETH paid is returned to the signer of the MultiSigWallet contract when the purchase fails. The ETH returned to the TribeOne contract is determined by the signer that calls the *submitTransaction* function of the MultiSigWallet contract. There may be cases where the ETH sent to the TribeOne contract does not match the actual amount refunded by the _agent address.

```
65        uint256 ii;
66        uint256 txIdx = txIds.current();
67        for (ii = 0; ii < sigLength; ii++) {
68            address _signer = _getSigner(_to, _value, _data, rs[ii], ss[ii], vs[ii]);
69            require(isSigner[_signer] && !isConfirmed[txIdx][_signer], "Not signer or duplicated signer for this transaction");
70            isConfirmed[txIdx][_signer] = true;
71        }
72        (bool success, ) = _to.call{value: _value}(_data);
73        require(success, "tx failed");
```

Figure 13 source code of *submitTransaction* function (Unfixed)

Fix recommendations: It is recommended that _agent addresses are refunded directly to the TribeOne contract when they are refunded.

Fix results: Acknowledged. The project party confirmed that the refunded ETH is sent from the _agent address to the signer address, which in turn sends it to the TribeOne contract via the MultiSigWallet contract.

[MSW-2 Info] Event trigger parameter error

Description: The txIdx used to trigger the SubmitTransaction event in the *submitTransaction* function of the MultiSigWallet contract is self-incrementing and does not correspond to this call.

```
60      function submitTransaction(
61          address _to,
62          uint256 _value,
63          bytes memory _data,
64          bytes32[] memory rs,
65          bytes32[] memory ss,
66          uint8[] memory vs
67      ) external payable onlySigner nonReentrant {
68          require(rs.length == ss.length && ss.length == vs.length, "Signaure lengths should be same");
69          uint256 sigLength = rs.length;
70          require(sigLength >= numConfirmationsRequired, "Less than needed required confirmations");
71          if (_value > 0) {
72              require(msg.value == _value, "Should send value");
73          }
74          uint256 ii;
75          uint256 txIdx = txIds.current();
76          for (ii = 0; ii < sigLength; ii++) {
77              address _signer = _getSigner(_to, _value, _data, rs[ii], ss[ii], vs[ii]);
78              require(isSigner[_signer] && !isConfirmed[txIdx][_signer], "Not signer or duplicated signer for this transaction");
79              isConfirmed[txIdx][_signer] = true;
80          }
81          (bool success, ) = _to.call{value: _value}(_data);
82          require(success, "tx failed");
83
84          txIds.increment();
85          emit SubmitTransaction(msg.sender, txIdx, _to, _value, _data);
86      }
```

Figure 14 source code of *submitTransaction* function (Unfixed)

Fix recommendations: It is recommended that the value of txIdx be increased after the event is triggered.

Fix results: Fixed.

```
50      function submitTransaction(
51          address _to,
52          uint256 _value,
53          bytes memory _data,
54          bytes32[] memory rs,
55          bytes32[] memory ss,
56          uint8[] memory vs
57      ) external payable onlySigner nonReentrant {
58          require(_to != address(0), "ZERO Address");
59          require(rs.length == ss.length && ss.length == vs.length, "Signaure lengths should be same");
60          uint256 sigLength = rs.length;
61          require(sigLength >= numConfirmationsRequired, "Less than needed required confirmations");
62          if (_value > 0) {
63              require(msg.value == _value, "Should send value");
64          }
65          uint256 ii;
66          uint256 txIdx = txIds.current();
67          for (ii = 0; ii < sigLength; ii++) {
68              address _signer = _getSigner(_to, _value, _data, rs[ii], ss[ii], vs[ii]);
69              require(isSigner[_signer] && !isConfirmed[txIdx][_signer], "Not signer or duplicated signer for this transaction");
70              isConfirmed[txIdx][_signer] = true;
71          }
72          (bool success, ) = _to.call{value: _value}(_data);
73          require(success, "tx failed");
74
75          emit SubmitTransaction(msg.sender, txIdx, _to, _value, _data);
76          txIds.increment();
77      }
```

Figure 15 source code of *submitTransaction* function (Fixed)

[MSW-3 Info] Lack of zero address checks

Description: The _to address in the *submitTransaction* function of the MultiSigWallet contract lacks a zero address check.

```
60    function submitTransaction(
61        address _to,
62        uint256 _value,
63        bytes memory _data,
64        bytes32[] memory rs,
65        bytes32[] memory ss,
66        uint8[] memory vs
67    ) external payable onlySigner nonReentrant {
68        require(rs.length == ss.length && ss.length == vs.length, "Signaure lengths should be same");
69        uint256 sigLength = rs.length;
70        require(sigLength >= numConfirmationsRequired, "Less than needed required confirmations");
71        if (_value > 0) {
72            require(msg.value == _value, "Should send value");
73        }
74        uint256 ii;
75        uint256 txIdx = txIds.current();
76        for (ii = 0; ii < sigLength; ii++) {
77            address _signer = _getSigner(_to, _value, _data, rs[ii], ss[ii], vs[ii]);
78            require(isSigner[_signer] && !isConfirmed[txIdx][_signer], "Not signer or duplicated signer for this transaction");
79            isConfirmed[txIdx][_signer] = true;
80        }
81        (bool success, ) = _to.call{value: _value}(_data);
82        require(success, "tx failed");
83
84        txIds.increment();
85        emit SubmitTransaction(msg.sender, txIdx, _to, _value, _data);
86    }
```

Figure 16 source code of *submitTransaction* function (Unfixed)

Fix recommendations: It is recommended to check whether the input address is a zero address.

Fix results: Fixed.

```
50    function submitTransaction(
51        address _to,
52        uint256 _value,
53        bytes memory _data,
54        bytes32[] memory rs,
55        bytes32[] memory ss,
56        uint8[] memory vs
57    ) external payable onlySigner nonReentrant {
58        require(_to != address(0), "ZERO Address");
59        require(rs.length == ss.length && ss.length == vs.length, "Signaure lengths should be same");
60        uint256 sigLength = rs.length;
61        require(sigLength >= numConfirmationsRequired, "Less than needed required confirmations");
62        if (_value > 0) {
63            require(msg.value == _value, "Should send value");
64        }
65        uint256 ii;
66        uint256 txIdx = txIds.current();
67        for (ii = 0; ii < sigLength; ii++) {
68            address _signer = _getSigner(_to, _value, _data, rs[ii], ss[ii], vs[ii]);
69            require(isSigner[_signer] && !isConfirmed[txIdx][_signer], "Not signer or duplicated signer for this transaction");
70            isConfirmed[txIdx][_signer] = true;
71        }
72        (bool success, ) = _to.call{value: _value}(_data);
73        require(success, "tx failed");
74
75        emit SubmitTransaction(msg.sender, txIdx, _to, _value, _data);
76        txIds.increment();
77    }
```

Figure 17 source code of *submitTransaction* function (Fixed)

[MSW-4 Info] Public function that could be declared as external

Description: Public functions that are never called by the contract should be declared external to save gas.

```
102     function getSigners() public view returns (address[] memory) {
103         return signers;
104     }
105
106     function getTransactionCount() public view returns (uint256) {
107         return txIds.current();
108     }
```

Figure 18 source code of *getSigners* and *getTransactionCount* functions (Unfixed)

Fix recommendations: It is recommended to modify the *getSigners* and *getTransactionCount* functions to external.

Fix results: Fixed.

```
93      function getSigners() external view returns (address[] memory) {
94          return signers;
95      }
96
97      function getTransactionCount() external view returns (uint256) {
98          return txIds.current();
99      }
```

Figure 19 source code of *getSigners* and *getTransactionCount* functions (Fixed)

[TBO-1 Medium] Repeatedly counting the number of late payments

Description: In the *_updatePenalty* function of the TribeOne contract, the overdue count is updated without determining whether the current overdue has been recorded. If the *setLoanDefaulted* function is called after the overdue date, it will update the overdue count once, and then the user will call the *payInstallment* function to make a repayment before the grace period, which will update the overdue count once again. The user's one overdue action is then recorded twice.

```
339     function payInstallment(uint256 _loanId, uint256 _amount) external payable nonReentrant {
340         Loan storage _loan = loans[_loanId];
341         require(_loan.status == Status.LOANACTIVED || _loan.status == Status.DEFAULTED, "TribeOne: Invalid status");
342         uint256 expectedNr = expectedNrOfPayments(_loanId);
343
344         address _loanCurrency = _loan.loanAsset.currency;
345         if (_loanCurrency == address(0)) {
346             _amount = msg.value;
347         }
348
349         uint256 paidAmount = _loan.paidAmount;
350         uint256 _totalDebt = totalDebt(_loanId); // loan + interest
351         {
352             uint256 expectedAmount = (_totalDebt * expectedNr) / _loan.loanRules.tenor;
353             require(paidAmount + _amount >= expectedAmount, "TribeOne: Insufficient Amount");
354             // out of rule, penalty
355             _updatePenalty(_loanId);
356         }
```

Figure 20 source code of *payInstallment* function

```
function _updatePenalty(uint256 _loanId) private {
    Loan storage _loan = loans[_loanId];
    require(_loan.status == Status.LOANACTIVED || _loan.status == Status.DEFAULTED, "TribeOne: Not actived loan");
    uint256 expectedNr = expectedNrOfPayments(_loanId);
    uint256 passedTenors = _loan.passedTenors;
    if (expectedNr > passedTenors) {
        _loan.nrOfPenalty += uint8(expectedNr - passedTenors);
    }
}
```

Figure 21 source code of *_updatePenalty* function

```
function setLoanDefaulted(uint256 _loanId) external nonReentrant {
    Loan storage _loan = loans[_loanId];
    require(_loan.status == Status.LOANACTIVED, "TribeOne: Invalid status");
    require(expectedLastPaymentTime(_loanId) < block.timestamp, "TribeOne: Not overdued date yet");

    _updatePenalty(_loanId);
    _loan.status = Status.DEFAULTED;

    emit LoanDefaulted(_loanId);
}
```

Figure 22 source code of *setLoanDefaulted* function (Unfixed)

Fix recommendations: It is recommended to remove the code related to updating overdue information in the *setLoanDefaulted* function.

Fix results: Fixed.

```
function setLoanDefaulted(uint256 _loanId) external nonReentrant {
    Loan storage _loan = loans[_loanId];
    require(_loan.status == Status.LOANACTIVED, "TribeOne: Invalid status");
    require(expectedLastPaymentTime(_loanId) < block.timestamp, "TribeOne: Not overdued date yet");

    _loan.status = Status.DEFAULTED;

    emit LoanDefaulted(_loanId);
}
```

Figure 23 source code of *setLoanDefaulted* function (Fixed)

[TBO-2 Medium] Uninitialised variables

Description: The admin in the TribeOne contract is used but not initialized.

```
if (_amount < _fundAmount) {
    _loan.status = Status.REJECTED;
    returnColleteral(_loanId);
    emit LoanRejected(_loanId, _agent);
} else {
    if (!isAdmin(msg.sender)) {
        require(IAssetManager(assetManager).isValidAutomaticLoan(_loan.loanAsset.currency, expectedPrice));
    }

    _loan.status = Status.APPROVED;
    address _token = _loan.loanAsset.currency;
```

Figure 24 source code of *approveLoan* function

Fix recommendations: It is recommended to add corresponding functions for adding or removing admin.

Fix results: Fixed. The *addAmin* and *removeAdmin* functions have been added.

```
98      function addAdmin(address _admin) external onlySuperOwner {
99          require(!isAdmin(_admin), "Already admin");
100         admins[_admin] = true;
101         emit AddAdmin(msg.sender, _admin);
102     }
103
104     function removeAdmin(address _admin) external onlySuperOwner {
105         require(isAdmin(_admin), "This address is not admin");
106         admins[_admin] = false;
107         emit RemoveAdmin(msg.sender, _admin);
108     }
```

Figure 25 source code of *andAdmin* and *removeAdmin* functions

[TBO-3 Medium] expectedLastPaymentTime function implementation flaws

Description: When the purchase is successful in the TribeOne contract, the result calculated by the *expectedLastPaymentTime* function is the successful purchase time.

```
438     function expectedLastPaymentTime(uint256 _loanId) public view returns (uint256) {
439         Loan storage _loan = loans[_loanId];
440         return _loan.loanStart + TENOR_UNIT * (_loan.passedTenors);
441     }
```

Figure 26 source code of *expectedLastPaymentTime* function (Unfixed)

Fix recommendations: It is recommended to confirm whether it should be loanStart+TENOR_UNIT.

Fix results: Fixed.

```
460     function expectedLastPaymentTime(uint256 _loanId) public view returns (uint256) {
461         Loan storage _loan = loans[_loanId];
462         return
463             _loan.passedTenors >= _loan.loanRules.tenor
464                 ? _loan.loanStart + TENOR_UNIT * (_loan.loanRules.tenor)
465                 : _loan.loanStart + TENOR_UNIT * (_loan.passedTenors + 1);
466     }
```

Figure 27 source code of *expectedLastPaymentTime* function (Fixed)

[TBO-4 Medium] Repeatedly sending tokens

Description: The *safeTransferFrom* function already sends the number of repayments to the assetManager contract when the repayment in the *payInstallment* function is a normal token, and the *collectInstallment* function duplicates the amount of repayments.

```
369             IAssetManager(assetManager).collectInstallment{value: _amount}(
370                 _loanCurrency,
371                 _amount,
372                 _loan.loanRules.interest,
373                 false
374             );
375         } else {
376             TribeOneHelper.safeTransferFrom(_loanCurrency, _msgSender(), assetManager, _amount);
377             IAssetManager(assetManager).collectInstallment(_loanCurrency, _amount, _loan.loanRules.interest, false);
378         }
```

Figure 28 source code of *payInstallment* function (Unfixed)

Fix recommendations: It is recommended to modify the *safeTransferFrom* function to use the TrbrOne contract as the target address.

Fix results: Fixed.

```
366         if (_loanCurrency == address(0)) {
367             if (dust > 0) {
368                 TribeOneHelper.safeTransferETH(_msgSender(), dust);
369             }
370             // TribeOneHelper.safeTransferETH(assetManager, _amount);
371             IAssetManager(assetManager).collectInstallment{value: _amount}(
372                 _loanCurrency,
373                 _amount,
374                 _loan.loanRules.interest,
375                 false
376             );
377         } else {
378             TribeOneHelper.safeTransferFrom(_loanCurrency, _msgSender(), address(this), _amount);
379             IAssetManager(assetManager).collectInstallment(_loanCurrency, _amount, _loan.loanRules.interest, false);
380         }
```

Figure 29 source code of *payInstallment* function (Fixed)

[TBO-5 Medium] Incorrect number of tokens returned

Description: When a purchase fails, the number of tokens returned to the _agent address is the number of user loans, not the amount actually paid for the purchase.

```
299             _loan.status = Status.LOANACTIVED;
300             _loan.loanStart = block.timestamp;
301             // user can not get back collateral in this case, we transfer collateral to AssetManager
302             address _currency = _loan.collateralAsset.currency;
303             uint256 _amount = _loan.collateralAsset.amount;
304             // TribeOneHelper.safeTransferAsset(_currency, assetManager, _amount);
```

Figure 30 source code of *relayNFT* function (Unfixed)

Fix recommendations: It is recommended that the value of _amount be changed to "_loan.loanAsset.amount + _loan.fundAmount".

Fix results: Fixed.

```
316             _loan.status = Status.FAILED;
317             // refund loan
318             // in the case when loan currency is ETH, loan amount should be fund back from agent to TribeOne AssetNanager
319             address _token = _loan.loanAsset.currency;
320             uint256 _amount = _loan.loanAsset.amount + _loan.fundAmount;
```

Figure 31 source code of *relayNFT* function (Fixed)

[TBO-6 Medium] Incorrect time judgement

Description: In the TribeOne contract, when calling the *lockRestAmount* function, the time judgement is problematic.

```
544     function lockRestAmount(uint256[] calldata _loanIds, address _currency) external nonReentrant {
545         uint256 len = _loanIds.length;
546         uint256 _amount = 0;
547         for (uint256 ii = 0; ii < len; ii++) {
548             uint256 _loanId = _loanIds[ii];
549             Loan storage _loan = loans[_loanId];
550             if (
551                 _loan.loanAsset.currency == _currency &&
552                 _loan.status == Status.POSTLIQUIDATION &&
553                 _loan.postTime + GRACE_PERIOD > block.timestamp
554             ) {
555                 _amount += _loan.restAmount;
556                 _loan.status = Status.RESTLOCKED;
557             }
558         }
559 
560         TribeOneHelper.safeTransferAsset(_currency, feeTo, _amount);
561     }
```

Figure 32 source code of *lockRestAmount* function (Unfixed)

Fix recommendations: It is recommended that "postTime + GRACE_PERIOD > block.timestamp" be changed to "postTime + GRACE_PERIOD <= block.timestamp".

Fix results: Fixed.

```
571     function lockRestAmount(uint256[] calldata _loanIds, address _currency) external nonReentrant {
572         uint256 len = _loanIds.length;
573         uint256 _amount = 0;
574         for (uint256 ii = 0; ii < len; ii++) {
575             uint256 _loanId = _loanIds[ii];
576             Loan storage _loan = loans[_loanId];
577             if (
578                 _loan.loanAsset.currency == _currency &&
579                 _loan.status == Status.POSTLIQUIDATION &&
580                 _loan.postTime + GRACE_PERIOD <= block.timestamp
581             ) {
582                 _amount += _loan.restAmount;
583                 _loan.status = Status.RESTLOCKED;
584             }
585         }
586 
587         TribeOneHelper.safeTransferAsset(_currency, feeTo, _amount);
588     }
```

Figure 33 source code of *lockRestAmount* function (Fixed)

[TBO-7 Medium] Incorrect calculation of late penalty fee

Description: The *getBackFund* function in the TribeOne contract does not count the number (nrOfPenalty) of late fees when calculating late fees.

```
512     function getBackFund(uint256 _loanId) external payable {
513         Loan storage _loan = loans[_loanId];
514         require(_msgSender() == _loan.borrower, "TribOne: Forbidden");
515         require(_loan.status == Status.POSTLIQUIDATION, "TribeOne: Invalid status");
516         require(_loan.postTime + GRACE_PERIOD > block.timestamp, "TribeOne: Time over");
517         uint256 _restAmount = _loan.restAmount;
518         require(_restAmount > 0, "TribeOne: No amount to give back");
519 
520         if (lateFee > 0) {
521             uint256 _amount = lateFee * (10**IERC20Metadata(feeCurrency).decimals()); // tenor late fee
522             TribeOneHelper.safeTransferFrom(feeCurrency, _msgSender(), address(this), _amount);
523         }
```

Figure 34 source code of *getBackFund* function (Unfixed)

Fix recommendations: It is recommended to confirm the business logic.

Fix results: Fixed.

```
542     function getBackFund(uint256 _loanId) external payable {
543         Loan storage _loan = loans[_loanId];
544         require(_msgSender() == _loan.borrower, "TribOne: Forbidden");
545         require(_loan.status == Status.POSTLIQUIDATION, "TribeOne: Invalid status");
546         require(_loan.postTime + GRACE_PERIOD > block.timestamp, "TribeOne: Time over");
547         uint256 _restAmount = _loan.restAmount;
548         require(_restAmount > 0, "TribeOne: No amount to give back");
549 
550         if (lateFee > 0) {
551             uint256 _amount = lateFee * (10**IERC20Metadata(feeCurrency).decimals()) * _loan.nrOfPenalty; // tenor late fee
552             TribeOneHelper.safeTransferFrom(feeCurrency, _msgSender(), address(this), _amount);
553         }
```

Figure 35 source code of *getBackFund* function (Fixed)

[TBO-8 Medium] Incorrect fee receiving address

Description: The lateFee in the *_withdrawNFT* function is sent to the feeTo address, while the lateFee in the *getBackFund* function is sent to the TribeOne contract. There is no function for withdrawing lateFee in TribeOne contracts.

```
512     function getBackFund(uint256 _loanId) external payable {
513         Loan storage _loan = loans[_loanId];
514         require(_msgSender() == _loan.borrower, "TribOne: Forbidden");
515         require(_loan.status == Status.POSTLIQUIDATION, "TribeOne: Invalid status");
516         require(_loan.postTime + GRACE_PERIOD > block.timestamp, "TribeOne: Time over");
517         uint256 _restAmount = _loan.restAmount;
518         require(_restAmount > 0, "TribeOne: No amount to give back");
519 
520         if (lateFee > 0) {
521             uint256 _amount = lateFee * (10**IERC20Metadata(feeCurrency).decimals()); // tenor late fee
522             TribeOneHelper.safeTransferFrom(feeCurrency, _msgSender(), address(this), _amount);
523         }
```

Figure 36 source code of *getBackFund* function (Unfixed)

Fix recommendations: It is recommended to modify it to send it directly to the feeTo address.

Fix results: Fixed.

```
544     function getBackFund(uint256 _loanId) external payable {
545         Loan storage _loan = loans[_loanId];
546         require(_msgSender() == _loan.borrower, "TribOne: Forbidden");
547         require(_loan.status == Status.POSTLIQUIDATION, "TribeOne: Invalid status");
548         require(_loan.postTime + GRACE_PERIOD > block.timestamp, "TribeOne: Time over");
549         uint256 _restAmount = _loan.restAmount;
550         require(_restAmount > 0, "TribeOne: No amount to give back");
551 
552         if (lateFee > 0) {
553             uint256 _amount = lateFee * (10**IERC20Metadata(feeCurrency).decimals()) * _loan.nrOfPenalty; // tenor late fee
554             TribeOneHelper.safeTransferFrom(feeCurrency, _msgSender(), address(feeTo), _amount);
555         }
```

Figure 37 source code of *getBackFund* function (Fixed)

[TBO-9 Medium] Incorrect token transfer

Description: As shown below, when the NFT purchase failed, _agent did not pass the tokens paid for the purchase (not ETH) directly into the TribeOne contract.

```
323             if (_token == address(0)) {
324                 require(msg.value >= _amount, "TribeOne: Less than loan amount");
325                 if (msg.value > _amount) {
326                     TribeOneHelper.safeTransferETH(_agent, msg.value - _amount);
327                 }
328                 // TribeOneHelper.safeTransferETH(assetManager, _amount);
329                 IAssetManager(assetManager).collectInstallment{value: _amount}(_token, _amount, _loan.loanRules.interest, true);
330             } else {
331                 // TribeOneHelper.safeTransferFrom(_token, _agent, assetManager, _amount);
332                 IAssetManager(assetManager).collectInstallment(_token, _amount, _loan.loanRules.interest, true);
333             }
```

Figure 36 source code of *relayNFT* function (Unfixed)

Fix recommendations: It is recommended to confirm if the tokens need to be transferred into the TribeOne contract first.

Fix results: Fixed.

```
321             address _token = _loan.loanAsset.currency;
322             uint256 _amount = _loan.loanAsset.amount + _loan.fundAmount;
323             if (_token == address(0)) {
324                 require(msg.value >= _amount, "TribeOne: Less than loan amount");
325                 if (msg.value > _amount) {
326                     TribeOneHelper.safeTransferETH(_agent, msg.value - _amount);
327                 }
328                 IAssetManager(assetManager).collectInstallment{value: _amount}(_token, _amount, _loan.loanRules.interest, true);
329             } else {
330                 TribeOneHelper.safeTransferFrom(_token, _agent, address(this), _amount);
331                 IAssetManager(assetManager).collectInstallment(_token, _amount, _loan.loanRules.interest, true);
332             }
```

Figure 37 source code of *relayNFT* function (Fixed)

[TBO-10 Medium] Incorrect penalty fee calculation

Description: In the *finalDebtAndPenalty* function, the contract needs to use the total amount of the user's loan to calculate the penalty fee, rather than using the user's unpaid portion of the loan.

```
533     function finalDebtAndPenalty(uint256 _loanId) public view returns (uint256) {
534         Loan storage _loan = loans[_loanId];
535         uint256 paidAmount = _loan.paidAmount;
536         uint256 _totalDebt = totalDebt(_loanId);
537         uint256 _penalty = (_loan.loanAsset.amount * penaltyFee) / 1000; // 5% penalty of loan amount
538         return _totalDebt + _penalty - paidAmount;
539     }
```

Figure 38 source code of *finalDebtAndPenalty* function (Unfixed)

Fix recommendations: It is recommended to use the user's unpaid portion of the loan to calculate the penalty fee.

Fix results: Fixed.

```
531     function finalDebtAndPenalty(uint256 _loanId) public view returns (uint256) {
532         Loan storage _loan = loans[_loanId];
533         uint256 paidAmount = _loan.paidAmount;
534         uint256 _totalDebt = totalDebt(_loanId);
535         uint256 _penalty = ((_totalDebt - paidAmount) * penaltyFee) / 1000; // 5% penalty of loan amount
536         return _totalDebt + _penalty - paidAmount;
537     }
```

Figure 39 source code of *finalDebtAndPenalty* function (Fixed)

[TBO-11 Low] Variable modification without cap

Description: The *setSettings* function is called to modify the rateFee and penaltyFee without setting the cap.

```
155     function setSettings(
156         address _feeTo,
157         uint256 _lateFee,
158         uint256 _penaltyFee,
159         address _salesManager,
160         address _assetManager
161     ) external onlyOwner {
162         require(_feeTo != address(0) && _salesManager != address(0) && _assetManager != address(0), "TribeOne: ZERO address");
163         feeTo = _feeTo;
164         lateFee = _lateFee;
165         penaltyFee = _penaltyFee;
166         salesManager = _salesManager;
167         assetManager = _assetManager;
168         emit SettingsUpdate(_feeTo, _lateFee, _penaltyFee, _salesManager, assetManager);
169     }
```

Figure 38 source code of *setSettings* function (Unfixed)

Fix recommendations: It is recommended to set a reasonable cap.

Fix results: Fixed.

```
155     function setSettings(
156         address _feeTo,
157         uint256 _lateFee,
158         uint256 _penaltyFee,
159         address _salesManager,
160         address _assetManager
161     ) external onlySuperOwner {
162         require(_feeTo != address(0) && _salesManager != address(0) && _assetManager != address(0), "TribeOne: ZERO address");
163         require(_lateFee <= 5 && penaltyFee <= 50, "TribeOne: Exceeded fee limit");
164         feeTo = _feeTo;
165         lateFee = _lateFee;
166         penaltyFee = _penaltyFee;
167         salesManager = _salesManager;
168         assetManager = _assetManager;
169         emit SettingsUpdate(_feeTo, _lateFee, _penaltyFee, _salesManager, assetManager);
170     }
```

Figure 39 source code of *setSettings* function (Fixed)

[TBO-12 Low] *createLoan* function implementation flaws

Description: The *createLoan* function of the TribeOne contract does not allow native tokens to be used as collateral tokens, and native tokens are added as collateral tokens in the AssetManager contract.

```
214         // Transfer Collateral from sender to contract
215         uint256 _fundAmount = _amounts[0];
216         uint256 _collateralAmount = _amounts[1];
217
218         // Transfer collateral to TribeOne
219         TribeOneHelper.safeTransferFrom(_collateralCurrency, _msgSender(), address(this), _collateralAmount);
220
221         loans[loanID].nftAddressArray = nftAddressArray;
222         loans[loanID].borrower = _msgSender();
223         loans[loanID].loanAsset = Asset({currency: _loanCurrency, amount: 0});
224         loans[loanID].collateralAsset = Asset({currency: _collateralCurrency, amount: _collateralAmount});
225         loans[loanID].loanRules = LoanRules({tenor: tenor, LTV: LTV, interest: interest});
226         loans[loanID].nftTokenIdArray = nftTokenIdArray;
227         loans[loanID].fundAmount = _fundAmount;
```

Figure 40 source code of *createLoan* function (Unfixed)

Fix recommendations: It is recommended to modify the relevant code and allow the use of ETH as a collateral token.

Fix results: Fixed.

```
205         // Transfer Collateral from sender to contract
206         uint256 _fundAmount = _amounts[0];
207         uint256 _collateralAmount = _amounts[1];
208
209         // Transfer collateral to TribeOne
210         if (_collateralCurrency == address(0)) {
211             require(msg.value >= _collateralAmount, "TribeOne: Insufficient collateral amount");
212             if (msg.value > _collateralAmount) {
213                 TribeOneHelper.safeTransferETH(msg.sender, msg.value - _collateralAmount);
214             }
215         } else {
216             TribeOneHelper.safeTransferFrom(_collateralCurrency, _msgSender(), address(this), _collateralAmount);
217         }
218
```

Figure 41 source code of *createLoan* function (Fixed)

[TBO-13 Low] Incorrect highest price determination

Description: In the TribeOne contract, when calling the *isValidAutomaticLoan* function to calculate whether the automatic loan condition is met, the quantity passed in by the *approveLoan* function should be _amount (i.e. the actual number of loans approved). If the expectedPrice is passed in, the _amount may not reach 200 USDC.

```
254         } else {
255             if (!isAdmin(msg.sender)) {
256                 require(IAssetManager(assetManager).isValidAutomaticLoan(_loan.loanAsset.currency, expectedPrice));
257             }
```

Figure 42 source code of *approveLoan* function (Unfixed)

Fix recommendations: It is recommended that the number passed in when calling the *isValidAutomaticLoan* function is _amount.

Fix results: Fixed.

```
252         } else {
253             if (!isAdmin(msg.sender)) {
254                 require(
255                     IAssetManager(assetManager).isValidAutomaticLoan(_loan.loanAsset.currency, _amount),
256                     "TribeOne: Exceeded loan limit"
257                 );
258             }
```

Figure 43 source code of *approveLoan* function (Fixed)

[TBO-14 Low] Lack of permission checks

Description: With the onlyOwner modifier removed, it may be called by a malicious address and when the third parameter is entered as false, it will be refunded regardless of whether the purchase was successful or not.

```
279     function relayNFT(
280         uint256 _loanId,
281         address _agent,
282         bool _accepted
283     ) external payable override nonReentrant {
284         Loan storage _loan = loans[_loanId];
285         require(_loan.status == Status.APPROVED, "TribeOne: Not approved loan");
286         require(_agent != address(0), "TribeOne: ZERO address");
287         if (_accepted) {
288             uint256 len = _loan.nftAddressArray.length;
289             for (uint256 ii = 0; ii < len; ii++) {
290                 TribeOneHelper.safeTransferNFT(
291                     _loan.nftAddressArray[ii],
292                     _agent,
293                     address(this),
294                     _loan.nftTokenTypeArray[ii],
295                     _loan.nftTokenIdArray[ii]
296                 );
297             }
```

Figure 44 source code of *relayNFT* function (Unfixed)

Fix recommendations: It is recommended that the onlyOwer modifier be added to restrict calling privileges.

Fix results: Fixed.

```
279     function relayNFT(
280         uint256 _loanId,
281         address _agent,
282         bool _accepted
283     ) external payable override onlyOwner nonReentrant {
284         Loan storage _loan = loans[_loanId];
285         require(_loan.status == Status.APPROVED, "TribeOne: Not approved loan");
286         require(_agent != address(0), "TribeOne: ZERO address");
287         if (_accepted) {
288             uint256 len = _loan.nftAddressArray.length;
289             for (uint256 ii = 0; ii < len; ii++) {
290                 TribeOneHelper.safeTransferNFT(
291                     _loan.nftAddressArray[ii],
292                     _agent,
293                     address(this),
294                     _loan.nftTokenTypeArray[ii],
295                     _loan.nftTokenIdArray[ii]
296                 );
297             }
```

Figure 45 source code of *relayNFT* function (Fixed)

[TBO-15 Low] Lack of collateral token value determination

Description: The fundAmount used to calculate the number of loans the user needs to repay is the one entered by the user when creating the loan. If the user enters a malicious amount, or if the value of the collateral is not equal to the fundAmount due to market fluctuations, this may result in the actual number of loans the user needs to repay not matching the actual one.

```
219         loans[loanID].nftAddressArray = nftAddressArray;
220         loans[loanID].borrower = _msgSender();
221         loans[loanID].loanAsset = Asset({currency: _loanCurrency, amount: 0});
222         loans[loanID].collateralAsset = Asset({currency: _collateralCurrency, amount: _collateralAmount});
223         loans[loanID].loanRules = LoanRules({tenor: tenor, LTV: LTV, interest: interest});
224         loans[loanID].nftTokenIdArray = nftTokenIdArray;
225         loans[loanID].fundAmount = _fundAmount;
226 
227         loans[loanID].status = Status.LISTED;
228         loans[loanID].nftTokenTypeArray = nftTokenTypeArray;
229 
230         emit LoanCreated(loanID, msg.sender);
```

Figure 46 source code of *createLoan* function (Unfixed)

```
260             _loan.status = Status.APPROVED;
261             address _token = _loan.loanAsset.currency;
262
263             _loan.loanAsset.amount = _amount - _loan.fundAmount;
264
265             if (_token == address(0)) {
266                 IAssetManager(assetManager).requestETH(_agent, _amount);
267             } else {
268                 IAssetManager(assetManager).requestToken(_agent, _token, _amount);
269             }
```

Figure 47 source code of *createLoan* function (Unfixed)

Fix recommendations: It is recommended that the value of collateral be judged at the time of loan creation and approval.

Fix results: Acknowledged. The the project party are checking if the loan is valid in the back-end side first and then only approving it. They have used the expectedPrice with some slippage value(5%) to accommodate for market fluctuations. In addition for the case when the loan price is over $200, they will approve the loan manually using their loan admin portal.

[TBO-16 Low] Failure to consider critical conditions when refusing a loan

Description: The loan should also be declined when the actual selling price of the NFT is equal to the fundAmount. In the case of equals, the number of borrowed tokens will be staged at 0, resulting in a user debt of 0. The *payInstallment* function will not be called properly and the user will not be able to withdraw the purchased NFT.

```
235     function approveLoan(
236         uint256 _loanId,
237         uint256 _amount,
238         address _agent
239     ) external override onlyOwner nonReentrant {
240         Loan storage _loan = loans[_loanId];
241         require(_loan.status == Status.LISTED, "TribeOne: Invalid request");
242         require(_agent != address(0), "TribeOne: ZERO address");
243
244         uint256 _fundAmount = _loan.fundAmount;
245         uint256 _LTV = _loan.loanRules.LTV;
246
247         uint256 expectedPrice = TribeOneHelper.getExpectedPrice(_fundAmount, _LTV, MAX_SLIPPAGE);
248         require(_amount <= expectedPrice, "TribeOne: Invalid amount");
249         // Loan should be rejected when requested loan amount is less than fund amount because of
250         if (_amount < _fundAmount) {
251             _loan.status = Status.REJECTED;
252             returnColleteral(_loanId);
253             emit LoanRejected(_loanId, _agent);
```

Figure 48 source code of *approveLoan* function (Unfixed)

Fix recommendations: It is recommended that "_amount < _fundAmount" be changed to "_amount <= _fundAmount".

Fix results: Fixed.

```
233     function approveLoan(
234         uint256 _loanId,
235         uint256 _amount,
236         address _agent
237     ) external override onlyOwner nonReentrant {
238         Loan storage _loan = loans[_loanId];
239         require(_loan.status == Status.LISTED, "TribeOne: Invalid request");
240         require(_agent != address(0), "TribeOne: ZERO address");
241 
242         uint256 _fundAmount = _loan.fundAmount;
243         uint256 _LTV = _loan.loanRules.LTV;
244 
245         uint256 expectedPrice = TribeOneHelper.getExpectedPrice(_fundAmount, _LTV, MAX_SLIPPAGE);
246         require(_amount <= expectedPrice, "TribeOne: Invalid amount");
247         // Loan should be rejected when requested loan amount is less than fund amount because of
248         if (_amount <= _fundAmount) {
249             _loan.status = Status.REJECTED;
250             returnColleteral(_loanId);
251             emit LoanRejected(_loanId, _agent);
```

Figure 49 source code of *approveLoan* function (Fixed)

[TBO-17 Low] Wrong modifier used

Description: The *setSettings* function uses the onlyOwner modifier and the admin address can also call the *setSettings* function without going through the MultiSigWallet contract.

```
155     function setSettings(
156         address _feeTo,
157         uint256 _lateFee,
158         uint256 _penaltyFee,
159         address _salesManager,
160         address _assetManager
161     ) external onlyOwner {
162         require(_feeTo != address(0) && _salesManager != address(0) && _assetManager != address(0), "TribeOne: ZERO address");
163         feeTo = _feeTo;
164         lateFee = _lateFee;
165         penaltyFee = _penaltyFee;
166         salesManager = _salesManager;
167         assetManager = _assetManager;
168         emit SettingsUpdate(_feeTo, _lateFee, _penaltyFee, _salesManager, assetManager);
169     }
```

Figure 50 source code of *setSettings* function (Unfixed)

Fix recommendations: It is recommended to use onlySuperOwner as a modifier to the *setSettings* function.

Fix results: Fixed.

```
155     function setSettings(
156         address _feeTo,
157         uint256 _lateFee,
158         uint256 _penaltyFee,
159         address _salesManager,
160         address _assetManager
161     ) external onlySuperOwner {
162         require(_feeTo != address(0) && _salesManager != address(0) && _assetManager != address(0), "TribeOne: ZERO address");
163         feeTo = _feeTo;
164         lateFee = _lateFee;
165         penaltyFee = _penaltyFee;
166         salesManager = _salesManager;
167         assetManager = _assetManager;
168         emit SettingsUpdate(_feeTo, _lateFee, _penaltyFee, _salesManager, assetManager);
169     }
```

Figure 51 source code of *setSettings* function (Fixed)

[TBO-18 Info] Trigger the wrong event

Description: In the Ownable contract, there is an error in the event triggering of the *removeAdmin* function.

```
104     function removeAdmin(address _admin) external onlySuperOwner {
105         require(isAdmin(_admin), "This address is not admin");
106         admins[_admin] = false;
107         emit AddAdmin(msg.sender, _admin);
108     }
```

Figure 52 source code of *removeAdmin* function (Unfixed)

Fix recommendations: It is recommended to change the event to RemoveAdmin.

Fix results: Fixed.

```
104     function removeAdmin(address _admin) external onlySuperOwner {
105         require(isAdmin(_admin), "This address is not admin");
106         admins[_admin] = false;
107         emit RemoveAdmin(msg.sender, _admin);
108     }
```

Figure 53 source code of *removeAdmin* function (Fixed)

[TBO-19 Info] Lack of length judgement for input parameters

Description: The *createLoan* function does not determine if the length of the _currencies array is 2.

```
174     function createLoan(
175         uint16[] calldata _loanRules, // tenor, LTV, interest, 10000 - 100% to use array - avoid stack too deep
176         address[] calldata _currencies, // _loanCurrency, _collateralCurrency, address(0) is native coin
177         address[] calldata nftAddressArray,
178         uint256[] calldata _amounts, // _fundAmount, _collateralAmount _fundAmount is the amount of _collateral in _loanAsset such as ETH
179         uint256[] calldata nftTokenIdArray,
180         TribeOneHelper.TokenType[] memory nftTokenTypeArray
181     ) external payable {
182         require(_loanRules.length == 3 && _amounts.length == 2, "TribeOne: Invalid parameter");
183         uint16 tenor = _loanRules[0];
184         uint16 LTV = _loanRules[1];
185         uint16 interest = _loanRules[2];
186         require(_loanRules[1] > 0, "TribeOne: LTV should not be ZERO");
187         require(_loanRules[0] > 0, "TribeOne: Loan must have at least 1 installment");
188         require(nftAddressArray.length > 0, "TribeOne: Loan must have at least 1 NFT");
189         address _collateralCurrency = _currencies[1];
190         address _loanCurrency = _currencies[0];
191         require(IAssetManager(assetManager).isAvailableLoanAsset(_loanCurrency), "TribeOne: Loan asset is not available");
```

Figure 54 source code of *createLoan* function (Unfixed)

Fix recommendations: It is recommended to determine if the length of the entered _currencies is 2.

Fix results: Fixed.

```
174     function createLoan(
175         uint16[] calldata _loanRules, // tenor, LTV, interest, 10000 - 100% to use array - avoid stack too deep
176         address[] calldata _currencies, // _loanCurrency, _collateralCurrency, address(0) is native coin
177         address[] calldata nftAddressArray,
178         uint256[] calldata _amounts, // _fundAmount, _collateralAmount _fundAmount is the amount of _collateral in _loanAsset such as ETH
179         uint256[] calldata nftTokenIdArray,
180         TribeOneHelper.TokenType[] memory nftTokenTypeArray
181     ) external payable {
182         require(_loanRules.length == 3 && _amounts.length == 2 && _currencies.length == 2, "TribeOne: Invalid parameter");
183         uint16 tenor = _loanRules[0];
184         uint16 LTV = _loanRules[1];
185         uint16 interest = _loanRules[2];
186         require(_loanRules[1] > 0, "TribeOne: LTV should not be ZERO");
187         require(_loanRules[0] > 0, "TribeOne: Loan must have at least 1 installment");
188         require(nftAddressArray.length > 0, "TribeOne: Loan must have at least 1 NFT");
189         address _collateralCurrency = _currencies[1];
190         address _loanCurrency = _currencies[0];
191         require(IAssetManager(assetManager).isAvailableLoanAsset(_loanCurrency), "TribeOne: Loan asset is not available");
192         require(
193             IAssetManager(assetManager).isAvailableCollateralAsset(_collateralCurrency),
194             "TribeOne: Collateral asset is not available"
195         );
```

Figure 55 source code of *createLoan* function (Fixed)

[TBO-20 Info] Test code not modified

Description: Some of the test code in the TribeOne contract has not been modified to the official version.

```
71      /**
72       * @dev It's for only testnet
73       * TODO It should reverted to above in mainnet
74       */
75      uint256 public TENOR_UNIT = 7 minutes;
76      uint256 public GRACE_PERIOD = 3 minutes;
77
```

Figure 56 source code of TrbieOne contract (Unfixed)

```
117     /**
118      * @dev It's just for only testnet.
119      * TODO It should be removed when mainnet deploy
120      */
121     function setPeriods(uint256 _tenorUnit, uint256 _gracePeriod) external {
122         require(msg.sender == address(0x6C641CE6A7216F12d28692f9d8b2BDcdE812eD2b));
123         TENOR_UNIT = _tenorUnit;
124         GRACE_PERIOD = _gracePeriod;
125     }
```

Figure 57 source code of TrbieOne contract (Fixed)

Fix recommendations: It is recommended that the code be modified to the official version.

Fix results: Fixed. The code is modified to the official version.

[TBO-21 Info] Lack of zero address checks

Description: The *setSettings* function of the TribeOne contract and the _assetManager in the constructor are not checked for zero address.

```
161
162     function setSettings(
163         address _feeTo,
164         uint256 _lateFee,
165         uint256 _penaltyFee,
166         address _salesManager,
167         address _assetManager
168     ) external onlyOwner {
169         require(_salesManager != address(0) && _feeTo != address(0), "TribeOne: ZERO address");
170         feeTo = _feeTo;
171         lateFee = _lateFee;
172         penaltyFee = _penaltyFee;
173         salesManager = _salesManager;
174         assetManager = _assetManager;
175         emit SettingsUpdate(_feeTo, _lateFee, _penaltyFee, _salesManager, assetManager);
176     }
177
```

Figure 58 source code of *setSettings* function (Unfixed)

```
97
98  constructor(
99      address _salesManager,
100     address _feeTo,
101     address _feeCurrency,
102     address _multiSigWallet,
103     address _assetManager
104 ) {
105     require(
106         _salesManager != address(0) && _feeTo != address(0) && _feeCurrency != address(0) && _multiSigWallet != address(0),
107         "TribeOne: ZERO address"
108     );
109     salesManager = _salesManager;
110     assetManager = _assetManager;
111     feeTo = _feeTo;
112     feeCurrency = _feeCurrency;
113
114     transferOwnership(_multiSigWallet);
115 }
```

Figure 59 source code of *constructor* function Unfixed)

Fix recommendations: It is recommended to check that the address entered is a zero address.

Fix results: Fixed.

```
97      constructor(
98          address _salesManager,
99          address _feeTo,
100         address _feeCurrency,
101         address _multiSigWallet,
102         address _assetManager
103     ) {
104         require(
105             _salesManager != address(0) &&
106                 _feeTo != address(0) &&
107                 _feeCurrency != address(0) &&
108                 _multiSigWallet != address(0) &&
109                 _assetManager != address(0),
110             "TribeOne: ZERO address"
111         );
112         salesManager = _salesManager;
113         assetManager = _assetManager;
114         feeTo = _feeTo;
115         feeCurrency = _feeCurrency;
116
117         transferOwnership(_multiSigWallet);
118     }
```

Figure 60 source code of *constructor* function (Fixed)

```
155     function setSettings(
156         address _feeTo,
157         uint256 _lateFee,
158         uint256 _penaltyFee,
159         address _salesManager,
160         address _assetManager
161     ) external onlyOwner {
162         require(_feeTo != address(0) && _salesManager != address(0) && _assetManager != address(0), "TribeOne: ZERO address");
163         feeTo = _feeTo;
164         lateFee = _lateFee;
165         penaltyFee = _penaltyFee;
166         salesManager = _salesManager;
167         assetManager = _assetManager;
168         emit SettingsUpdate(_feeTo, _lateFee, _penaltyFee, _salesManager, assetManager);
169     }
```

Figure 61 source code of *setSettings* function (Fixed)

Other Audit Items Descriptions

1. About the centralisation risk

AssetManage contracts are currently owned by EOA, and our team recommends that project party use DAO, TimeLock contracts and multi-signature wallets as contract owners to reduce the risk of centralisation. Their response is as follows:

For functions like listing loan, relay loan, and dispersing funds from the contract to buy an NFT the project party has implemented multi-sig owner wallets - reducing the potential of exploitation. However they are also developing a DAO contract for handling all the admin activities mentioned above like *requestETH*, etc. Once its setup all ownership will be shifted accordingly. Till then they have kept the functionalities with multi-sig owner wallet access while keeping the admin key very safe behind very secure kubectl environments with database vaults.

Note: Even if the MultiSigWallet contract is used as the owner, both the admin and the owner can call the *relayNFT* and *approveLoan* functions. If the *approveLoan* function is called by admin, there is no limit to the 200 USDC loan amount.

2. Business logic related notes

When a user creates a loan, a certain number of collateral tokens are sent to the TribeOne contract. Whether the loan is approved or not is determined off-chain by the project owner. Before a loan is approved, the user can cancel it themselves and get their collateral tokens back.

If the loan is approved, the tokens needed to purchase the NFT are sent directly from the AssetManage contract to the _agent address. The _agent address is entered by the owner at the time of the call. If the approval fails, the user's collateral tokens will be returned to the user.

If the purchase is successful, the contract owner will send the NFT from the _agent address to the TribeOne contract. The user's collateral tokens are then sent to the AssetManage contract, after which the user cannot retrieve the collateral tokens.

If the purchase fails, _agent will return the funds to the TribeOne contract. If the token paid for the purchase is ETH, it is sent from the _agent address to the signer address of the MultiSigWallet contract, which in turn sends it to the Tribe One contract via the *submitTransaction* function of the MultiSigWallet contract. There may be cases where the signer address is mischievous, i.e. it receives a refund and sends little or no ETH to the TribeOne contract.

The user pays off the loan within the specified time to receive the purchased NFT tokens. If there is an overdue behavior, you cannot collect it directly after paying off the loan, you need to manually call the

withdrawNFT function to collect it. If overdue for more than a certain period of time, NFT tokens may be sent to the salesManager, who will then sell them and send the funds back to the TribeOne contract to repay the user's loan. The salesManager is an external contract or address and is not included in the scope of this audit, there may be cases where the funds sent by salesManager do not match the actual price sold.

Appendix 1 Vulnerability Severity Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

Appendix 2 Description of Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1. Compiler Version Security

The old version of the compiler may cause various known security issues. Developers are advised to specify the contract code to use the latest compiler version and eliminate the compiler alerts.

1.2. Deprecated Items

The Solidity smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, such as throw, years, etc. To eliminate the potential pitfalls they may cause, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3. Redundant Code

Redundant code in smart contracts can reduce code readability and may require more gas consumption for contract deployment. It is recommended to eliminate redundant code.

1.4. SafeMath Features

Check whether the functions within the SafeMath library are correctly used in the contract to perform mathematical operations, or perform other overflow prevention checks.

1.5. require/assert Usage

Solidity uses state recovery exceptions to handle errors. This mechanism will undo all changes made to the state in the current call (and all its subcalls) and flag the errors to the caller. The functions assert and require can be used to check conditions and throw exceptions when the conditions are not met. The assert function can only be used to test for internal errors and check non-variables. The require function is used to confirm the validity of conditions, such as whether the input variables or contract state variables meet the conditions, or to verify the return value of external contract calls.

1.6. Gas Consumption

The smart contract virtual machine needs gas to execute the contract code. When the gas is insufficient, the code execution will throw an out of gas exception and cancel all state changes. Contract developers are required to control the gas consumption of the code to avoid function execution failures due to insufficient gas.

1.7. Visibility Specifiers

Check whether the visibility conforms to design requirement.

1.8. Fallback Usage

Check whether the Fallback function has been used correctly in the current contract.

2. General Vulnerability

2.1. Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Solidity can handle up to 256-bit numbers (2**256-1). If the maximum number is increased by 1, it will overflow to 0. Similarly, when the number is a uint type, 0 minus 1 will underflow to get the maximum number value. Overflow conditions can lead to incorrect results, especially if its possible results are not

expected, which may affect the reliability and safety of the program. For the compiler version after Solidity 0.8.0, smart contracts will perform overflow checking on mathematical operations by default. In the previous compiler versions, developers need to add their own overflow checking code, and SafeMath library is recommended to use.

2.2. Reentrancy

The reentrancy vulnerability is the most typical Ethereum smart contract vulnerability, which has caused the DAO to be attacked. The risk of reentry attack exists when there is an error in the logical order of calling the call.value() function to send assets.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. In solidity, it is common to use block information as a random factor to generate, but such use is insecure. Block information can be controlled by miners or obtained by attackers during transactions, and such random numbers are to some extent predictable or collidable.

2.4. Transaction-Ordering Dependence

In the process of transaction packing and execution, when faced with transactions of the same difficulty, miners tend to choose the one with higher gas cost to be packed first, so users can specify a higher gas cost to have their transactions packed and executed first.

2.5. DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state. There are various reasons for the denial of service of a smart contract, including malicious revert when acting as the recipient of a transaction, gas exhaustion caused by code design flaws, etc.

2.6. Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.7. call/delegatecall Security

Solidity provides the call/delegatecall function for function calls, which can cause call injection vulnerability if not used properly. For example, the parameters of the call, if controllable, can control this contract to perform unauthorized operations or call dangerous functions of other contracts.

2.8. Returned Value Security

In Solidity, there are transfer(), send(), call.value() and other methods. The transaction will be rolled back if the transfer fails, while send and call.value will return false if the transfer fails. If the return is not correctly

judged, the unanticipated logic may be executed. In addition, in the implementation of the transfer/transferFrom function of the token contract, it is also necessary to avoid the transfer failure and return false, so as not to create fake recharge loopholes.

2.9. tx.origin Usage

The tx.origin represents the address of the initial creator of the transaction. If tx.origin is used for permission judgment, errors may occur; in addition, if the contract needs to determine whether the caller is the contract address, then tx.origin should be used instead of extcodesize.

2.10. Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.11. Overriding Variables

There are complex variable types in Solidity, such as structures, dynamic arrays, etc. When using a lower version of the compiler, improperly assigning values to it may result in overwriting the values of existing state variables, causing logical exceptions during contract execution.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

Appendix 4 About Beosin

BEOSIN is a leading global blockchain security company dedicated to the construction of blockchain security ecology, with team members coming from professors, post-docs, PhDs from renowned universities and elites from head Internet enterprises who have been engaged in information security industry for many years. BEOSIN has established in-depth cooperation with more than 100 global blockchain head enterprises; and has provided security audit and defense deployment services for more than 1,000 smart contracts, more than 50 blockchain platforms and landing application systems, and nearly 100 digital financial enterprises worldwide. Relying on technical advantages, BEOSIN has applied for nearly 50 software invention patents and copyrights.

BEOSIN

Blockchain Security

Official Website

https://beosin.com

Twitter

https://twitter.com/Beosin_com