Smart contract security audit report

Caretta INU smart contract security audit

Audit Team : Noneage security team

Audit Date : Dec. 18 , 2022

Caretta INU Smart Contract Security Audit Report

1. Overview

On Dec 18, 2022, the security team of Noneage Technology received the security audit request of the **Caretta INU project**. The team completed the **Caretta INU smart contract** security audit on December 18. the security audit experts of Noneage Technology communicate with the relevant interface people of the **Caretta INU** project, maintain information symmetry, conduct security audits under controllable operational risks, and try to avoid project generation and operation during the test process. Cause risks.

Through communicat and feedback with **Caretta INU** project party, it is confirmed that the loopholes and risks found in the audit process have been repaired or within the acceptable range. The result of this **Caretta INU** smart contract security audit: **passed.**

Audit Report MD5 ： 3A55D71CB2CDE2B2C9BA958FD8C9B232

2. Background

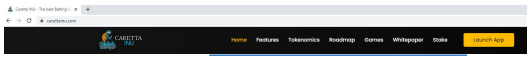
2.1 Project Description

Project name: CARETTA INU

Contract type: DeFi Token

contract Code language: Solidity

Project website : https://carettainu.com



Contract documents: CarettaINU.sol

```
/**
 *Submitted for verification at Etherscan.io on 2022-12-13
*/

/**
*/

// CARETTA INU

// Telegram : @carettainu
// Website : www.carettainu.com
// Twitter : twitter.com/carettainutoken
// BEWARE OF SCAMMERS

// SPDX-License-Identifier: Unlicensed
pragma solidity ^0.8.4;
```

2.2 Audit Range

The contract file and corresponding MD5 provided by Caretta INU :

Caretta INU.sol	70F89CCD31265F919DAFC0823A109F89

2.3 Security Audit List

The security experts of Noneage Technology conduct security audits on the security audit list within the agreement, The scope of this smart contract security audit does not include new attack methods that may appear in the future, does not include the code after contract upgrades or tampering, and is not included in the subsequent cross-country, does not include cross-chain deployment, does not include project front-end code security and project platform server security.

This smart contract security audit list includes the following:

- Integer overflow
- Reentry attack
- Floating point numbers and numerical precision
- Default visibility
- Tx.origin authentication
- Wrong constructor Return value not verified
- Insecure random numbers
- Timestamp dependency
- Transaction order is dependent
- Delegatecall
- Call
- Denial of service Logic design flaws
- Fake recharge vulnerability Short address attack Uninitialized storage pointer Additional token
- issuance Frozen account bypass
- Access control
- Gas usage

3. Contract Structure Analysis

3.1 Directory Structure

```
├—CARETTA INU
│ CarettaINU.sol
```

3.2 CARETTA INU contract

Contract

CarettaINU.sol

mint(address account, uint256
amount) addMiner(address account)
removeMiner(address account)
delegates(address delegator)
delegate(address delegatee)
delegateBySig(address delegatee,uint nonce,uint expiry,uint8 v,bytes32 r,bytes
32 s) getCurrentVotes(address account)

getPriorVotes(address account, uint

blockNumber) _delegate(address delegator,

address delegatee)

_moveDelegates(address srcRep, address dstRep, uint256 amount)
_writeCheckpoint(address delegatee,uint32 nCheckpoints,uint256 oldVotes,uint256
newVotes) safe32(uint n, string memory errorMessage) getChainId()

pendingReward(uint256 _pid, address _user)
massUpdatePools()
updatePool(uint256 _pid,uint256 _amount,bool isAdd)
deposit(uint256 _pid, uint256 _amount) depositLend(PoolInfo
memory pool,uint256 _amount) withdrawLend(PoolInfo
memory pool,uint256 _amount) withdraw(uint256 _pid,
uint256 _amount)
safeLpTransfer(PoolInfo memory pool,address _to, uint256 _amount)
emergencyWithdraw(uint256 _pid)
safeTransfer(address _to, uint256 _amount) dev(
address _devaddr)
operation(address _opaddr)
fund(address _fundaddr)
setFee(address _feeaddr)

poolLength() setBaseReward(uint256
_base)
setRITPerBlock(uint256 _RITPerBlock)
setFeebase(uint256 _feeBase)
setFee(uint256 _fee) GetPoolInfo(uint256
id) GetURITInfo(uint256 id,address addr)

add(uint256 _pid,uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate,uint256
_min,uint256 _max,uint256 _deposit_fee,uint256 _withdraw_fee,ERC20 _rewardToken) set(uint256
_pid, uint256 _allocPoint, bool _withUpdate,uint256 _min,uint256 _max,uint256
_deposit_fee,uint256 _withdraw_fee)
getMultiplier(uint256 _from, uint256 _to)
pending(uint256 _pid, address _uRIT)
rewardLp(uint256 _pid, address _user)
allRewardLp(uint256 _pid) massUpdatePools()
updatePool(uint256_pid,uint256_amount,boolisAdd)
deposit(uint256 _pid, uint256 _amount)
safeWithdraw(uint256 _pid)
withdraw(uint256 _pid, uint256 _amount)
safeLpTransfer(uint256 _pid,address _to, uint256 _amount)
approve(PoolInfo memory pool)
calcProfit(uint256 _pid)
futou(PoolInfo memory pool)
harvest(uint256 _pid)
safeRITTransfer(address _to, uint256 _amount) dev(address
_devaddr)

setFeeAddr(address _feeaddr

poolLength() setBaseReward(uint256
_base)
setRITPerBlock(uint256 _RITPerBlock)
setFeebase(uint256 _feeBase)
setFee(uint256 _fee)
GetPoolInfo(uint256id)

pendingReward(uint256 _pid, address _user)
massUpdatePools()
updatePool(uint256 _pid,uint256 _amount,bool isAdd)
deposit(uint256 _pid, uint256 _amount) depositLend(PoolInfo
memory pool,uint256 _amount) withdrawLend(PoolInfo
memory pool,uint256 _amount) withdraw(uint256 _pid,
uint256 _amount)
safeLpTransfer(PoolInfo memory pool,address _to, uint256 _amount)
emergencyWithdraw(uint256 _pid)
safeTransfer(address _to, uint256 _amount) dev(
address _devaddr)
operation(address _opaddr)
fund(address _fundaddr)
setFee(address _feeaddr)

poolLength() setBaseReward(uint256
_base)
setRITPerBlock(uint256 _RITPerBlock)
setFeebase(uint256 _feeBase)
setFee(uint256 _fee) GetPoolInfo(uint256
id) GetURITInfo(uint256 id,address addr)
add(uint256 _pid,uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate,uint256
_min,uint256 _max,uint256 _deposit_fee,uint256 _withdraw_fee,ERC20 _rewardToken) set(uint256
_pid, uint256 _allocPoint, bool _withUpdate,uint256 _min,uint256 _max,uint256
_deposit_fee,uint256 _withdraw_fee)
getMultiplier(uint256 _from, uint256 _to)
pending(uint256 _pid, address _uRIT)
rewardLp(uint256 _pid, address _user)
allRewardLp(uint256 _pid) massUpdatePools()
updatePool(uint256_pid,uint256_amount,boolisAdd)
deposit(uint256 _pid, uint256 _amount)
safeWithdraw(uint256 _pid)
withdraw(uint256 _pid, uint256 _amount)
safeLpTransfer(uint256 _pid,address _to, uint256 _amount)
approve(PoolInfo memory pool)
calcProfit(uint256 _pid)
futou(PoolInfo memory pool)
harvest(uint256 _pid)
safeRITTransfer(address _to, uint256 _amount) dev(address
_devaddr)
setFeeAddr(address _feeaddr

)

poolLength() setBaseReward(uint256
_base)
setRITPerBlock(uint256 _RITPerBlock)
setFeebase(uint256 _feeBase)
setFee(uint256 _fee)
GetPoolInfo(uint256id)

GetURITInfo(uint256 id,address addr)

add(ILHB _kswap,uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate,uint256 _min,uint256 _max,uint256 _deposit_fee,uint256 _withdraw_fee,IERC20 _rewardToken) set(uint256 _pid, uint256 _allocPoint, bool _withUpdate,uint256 _min,uint256 _max,uint256 _deposit_fee,uint256 _withdraw_fee,IERC20 _rewardToken)
getMultiplier(uint256 _from, uint256 _to)
pending(uint256 _pid, address _uRIT)
balanceOfUnderlying(PoolInfo memory pool)
rewardLp(uint256 _pid, address _user)
allRewardLp(uint256 _pid) massUpdatePools()
updatePool(uint256_pid,uint256_amount,boolisAdd)
approve(PoolInfo memory pool)
deposit(uint256 _pid, uint256 _amount)
safeWithdraw(uint256 _pid)
withdraw(uint256 _pid, uint256 _amount)
safeLpTransfer(uint256 _pid,address _to, uint256 _min)
calcProfit(uint256 _pid)
futou(PoolInfo memory pool)
harvest(uint256 _pid)
safeRITTransfer(address _to, uint256 _amount) dev(address _devaddr)
setFeeAddr(address _feeaddr)

setCompleted(uint256 completed)

setPause(uint256 _pause)

executeTransaction(address target, uint value, string memory signature, bytes memory data)

_burn(uint256 _pid,uint256 _shares,address _user)
_mint(uint256 _pid,uint256 _amount,address _user,uint256 _allBalance) getWithdrawBalance(uint256 _pid,uint256 _shares,uint256 _allBalance) getWithdrawShares(uint256 _pid,uint256 _amount,address _user,uint256 _userBalance) userSharesReward(uint256 _pid,address _user,uint256 _allReward)
addRouter(address a,address b)
removeRouter(address a)
swap(IUniswapV2Router02 router,address token0,address token1,uint256 input) initRouters()

marketListSize() setCollateralFactor(uint factor) setLiquidationFactor(uint factor)
setPrice(address market, uint price)
pairFor(MarketInterface market)
getDecimal(address t)
getPrice(MarketInterface market)
addMarket(address market)
getAccountLiquidity(address account)
getAccountHealth(address account)

calculateHealthIndex(uint supplyValue, uint borrowValue)

getAccountValues(address account)
liquidateCollateral(address borrower, address liquidator, uint amount, MarketInterface collateralMarket)

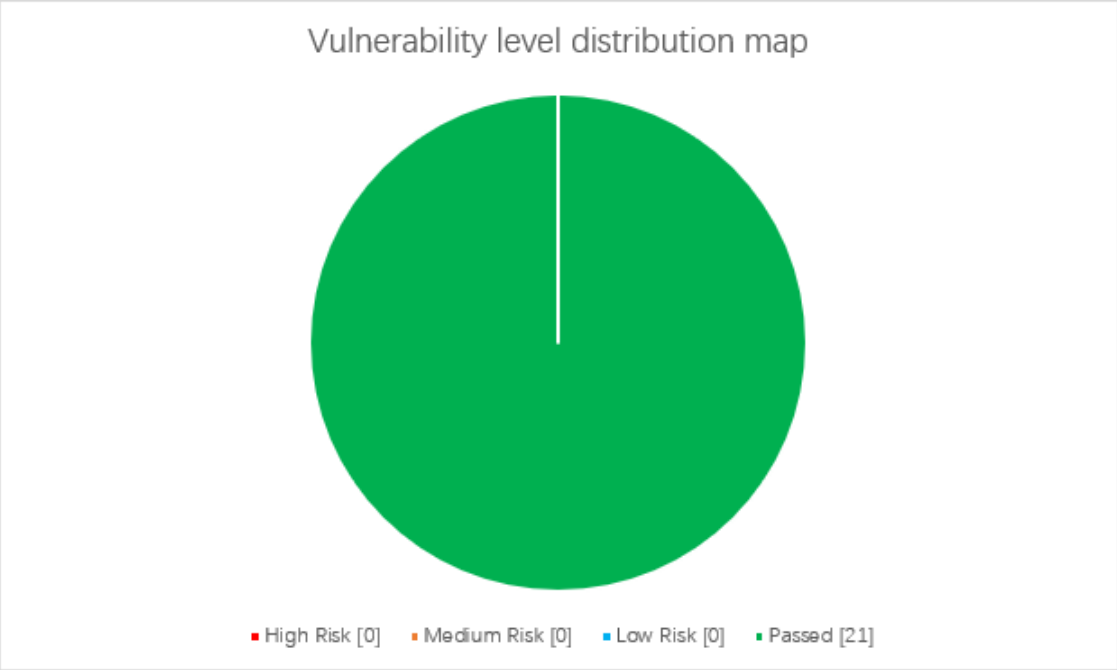
setUtilizationRateFraction(uint256 _utilizationRateFraction) setFutou(address _futou)
getCash()
utilizationRate(uint256 cash, uint256 borrowed, uint256 reserves)
getBorrowRate(uint256 cash, uint256 borrowed, uint256 reserves)
getSupplyRate(uint256 cash, uint256 borrowed, uint256 reserves)
borrowRatePerBlock()
supplyRatePerBlock() supplyOf(address user) borrowBy(address user)
updatedBorrowBy(address user)
updatedSupplyOf(address user)
setController(Controller _controller)
supply(uint256 amount)
supplyInternal(address supplier, uint256 amount)
redeem(uint256 amount)
redeemInternal(address supplier, address receiver, uint256 amount)
borrow(uint256 amount)
accrueInterest()
calculateBorrowDataAtBlock(uint256 newBlockNumber)
calculateSupplyDataAtBlock(uint256 newBlockNumber)
getUpdatedTotalBorrows()
getUpdatedTotalSupply()
payBorrow(uint256 amount)
payBorrowInternal(address payer, address borrower, uint256 amount)
liquidateBorrow(address borrower, uint256 amount, MarketInterface collateralMarket)
transferTo(address sender, address receiver, uint256 amount)

4. Audit Details

4.1 Vulnerabilities Distribution

Vulnerabilities in this security audit are distributed by risk level, as follows：

Vulnerability level distribution			
High risk	Medium risk	Low risk	Passed
0	0	0	21



This smart contract security audit has 0 high-risk vulnerabilities, 0 medium-risk vulnerabilities, 0 low-risk vulnerabilities, and 21 passed, with a high security level.

4.2 Vulnerabilities Details

A security audit was conducted on the smart contract within the agreement, and no security vulnerabilities that could be directly exploited and generated security problems were found, and the security audit was passed.

4.3 Other Risks

Other risks refer to the code that contract security auditors consider to be risky, which may affect the stability of the project under certain circumstances, but cannot constitute a security issue that directly endangers the security.

4.3.1 Function attribute problem

- **Question detail**

In the CARETTAINU contract, the setUtilizationRateFraction() method has the function of setting variables. Since this method is a public attribute, whether there is any address call here, causing other functions of the project to be unstable.

```
•   function setUtilizationRateFraction(uint256 _utilizationRateFraction)
public{
        utilizationRateFraction = _utilizationRateFraction;
    }
```

Safety advice

The above method is recommended to add an administrator decorator.

- **Update status**

Through communication with the CARETTAINU team, Safety advice has been adopted.

4.3.2 Redundant code

- **Question detail**

The CARETTAINU contract defines the futou variable and the setFutou() method, but the variable and function are not used in the contract code.

```
address public futou;
function setFutou(address _futou) public{
    futou = _futou;
}
```

Safety advice

It is recommended to remove the setFutou() method and futou variable definition.

- **Update status**

Through communication with the CARETTAINU team, Safety advice has been adopted.

4.3.3 Duplicate function

- **Question detail**

In the CARETTAINU contract, therearesupplyOf()methodand balanceOf() method, as well as supply() method and mint() method, but the internal functions of the above two methods are the same.

```
function supplyOf(address user) public view returns (uint256) {
    return supplies[user].supply;
}
function balanceOf(address user) public view returns (uint256) {
    return supplies[user].supply;
}

function supply(uint256 amount) public {
    supplyInternal(msg.sender, amount);
    emit Supply(msg.sender, amount);
}
function mint(uint256 amount) public {
    supplyInternal(msg.sender, amount);
    emit Supply(msg.sender, amount);
}
```

Safety advice

It is recommended to remove one of the above two methods.

- **Update status**

Through communication with the CARETTAINU team, Safety advice has been adopted.

4.3.4 Address validity check

- **Question detail**

In the CARETTAINU contract, multiple methods of address input are not validated, and there may be empty addresses. As follows:

```
    function updatedBorrowBy(address user) public view returns (uint256) {
        BorrowSnapshot storage snapshot = borrows[user];
        if (snapshot.principal == 0)
            return 0;
        uint256 newTotalBorrows;
        uint256 newBorrowIndex;
        (newTotalBorrows, newBorrowIndex) =
calculateBorrowDataAtBlock(block.number);
        return
snapshot.principal.mul(newBorrowIndex).div(snapshot.interestIndex);
    }
    function updatedSupplyOf(address user) public view returns (uint256) {
        SupplySnapshot storage snapshot = supplies[user];
        if (snapshot.supply == 0)
            return 0;
        uint256 newTotalSupply;
        uint256 newSupplyIndex;
        (newTotalSupply, newSupplyIndex) =
calculateSupplyDataAtBlock(block.number);
        return snapshot.supply.mul(newSupplyIndex).div(snapshot.interestIndex);
    }
```

- **Safety advice**

It is recommended to add a verification code for checking address validity, as shown below:

```
require(user != address(0) , "ADDRESS ERROR!!!");
```

Update status

Through communication with the CARETTAINU team, Safety advice has been adopted.

4.3.5 Administrator authority is too large

- **Question detail**

In the CARETTAINU contract, the administrator onlyOwner can set multiple values. If the administrator is manipulated by malicious personnel, it may cause abnormal capital loss and shake market stability, as shown in the following code:

```
    function setCollateralFactor(uint factor) public onlyOwner {
        collateralFactor = factor;
    }
    function setLiquidationFactor(uint factor) public onlyOwner {
        liquidationFactor = factor;
    }
    function setPrice(address market, uint price) public onlyOwner {
        require(markets[market]);
        prices[market] = price;
    }
    function addMarket(address market) public onlyOwner {
        address marketToken = MarketInterface(market).token();
        require(marketsByToken[marketToken] == address(0));
        markets[market] = true;
        marketsByToken[marketToken] = market;
        marketList.push(market);
    }
```

- **Safety advice**

On the premise of ensuring security, keep multiple copies of the private key reasonably.

- **Update status**

Through communication with the CARETTAINU team, multi-signature address control has been used.

4.3.6 Obtain minting operations

- **Question detail**

In the CARETTAINU contract, there is an operation to obtain minting funds. In the code shown below, the harvest() method is a public attribute. The calling process here is: harvest()——calcProfit()—— futou(). We mainly look at the key operations during the execution of this process. The harvest() method only needs to enter the value of pid and meet (ba<=0) and (pool.lpSupply<=0), then this series of calling processes can be run. In the end, pool.thirdPool.mint(ba); minting will be executed. The minting method here only passes in the value, not the address, so the default minting address is the address of the caller of msg.sender. In other words, as long as there is a pid value that meets the conditions, anyone can perform the minting operation of the process.

```
    function harvest(uint256 _pid) public {
        calcProfit(_pid);
        emit ReInvest(_pid);
    }
    function calcProfit(uint256 _pid) private{
        PoolInfo storage pool = poolInfo[_pid];
        address[] memory cTokens = new address[](1);
        cTokens[0] = address(pool.thirdPool);

 ILHB(0x6537d6307ca40231939985BCF7D83096Dd1B4C09).claimComp(address(this),
cTokens);
        pool.thirdPool.redeem(0);
        uint256 ba = pool.rewardToken.balanceOf(address(this));
        if(ba > baseReward){
            uint256 profitFee = ba.mul(fee).div(feeBase);
            pool.rewardToken.safeTransfer(feeaddr,profitFee);
            ba = ba.sub(profitFee);
            swap(router, address(pool.rewardToken),address(pool.lpToken), ba);
        }
        futou(pool);
    }
    function futou(PoolInfo memory pool) private {
        uint256 ba = pool.lpToken.balanceOf(address(this));
        if(ba<=0){
            return;
        }
        if(pool.lpSupply<=0){
            pool.lpToken.transfer(feeaddr,ba);
            return;
        }
        pool.thirdPool.mint(ba);
    }
```

- **Safety advice**

It is recommended to set the attribute of the harvest() method to internal.

- **Update status**

Through communication with the CARETTAINU team, Safety advice has been adopted.

4.3.7 Validity and safety of transfer value

- **Question detail**

In the CARETTAINU contract, the transfer() interface does not verify the validity of the input transfer value, as shown in the following code, which uses rit.transfer(_to, RITBal); in this line of code, when rit.balanceOf(address(this)) is zero and RITBaal is zero. The possibility that the value of RITBaal is zero cannot be ruled out by judgment.

```
function safeRITTransfer(address _to, uint256 _amount) internal {
    uint256 RITBal = rit.balanceOf(address(this));
    if (_amount > RITBal) {
        rit.transfer(_to, RITBal);
    } else {
        rit.transfer(_to, _amount);
    }
}
```

Safety advice

It is recommended to add a judgment condition that the input value is greater than 0 before the transfer() interface.

- **Update status**

Through communication with the CARETTAINU team, Safety advice has been adopted.

5. Security Audit Tool

Tool name	Tool Features
Oyente	Can be used to detect common bugs in smart contracts
securify	Common types of smart contracts that can be verified
MAIAN	Multiple smart contract vulnerabilities can be found and classified
Noneage Internal Toolkit	Noneage(hawkeye system) self-developed toolkit + https://audit.noneage.com

6. Vulnerability assessment criteria

Vulnerability level	Vulnerability description
High risk	Vulnerabilities that can directly lead to the loss of contracts or users' digital assets, such as integer overflow vulnerabilities, false recharge vulnerabilities, re-entry vulnerabilities, illegal token issuance, etc. Vulnerabilities that can directly cause the ownership change of the token contract or verification bypass, such as: permission verification bypass, call code injection, variable coverage, unverified return value, etc. Vulnerabilities that can directly cause the token to work normally, such as denial of service vulnerabilities, insecure random numbers, etc.
Medium risk	Vulnerabilities that require certain conditions to trigger, such as vulnerabilities triggered by the token owner's high authority, and transaction sequence dependent vulnerabilities. Vulnerabilities that cannot directly cause asset loss, such as function default visibility errors, logic design flaws, etc.
Low risk	Vulnerabilities that are difficult to trigger, or vulnerabilities that cannot lead to asset loss, such as vulnerabilities that need to be triggered at a cost higher than the benefit of the attack, cannot lead to incorrect coding of security vulnerabilities.

NONEAGE

Telephone：86-17391945345 18511993344

Email ：support@noneage.com

Site ：www.noneage.com

Weibo ：weibo.com/noneage