BEOSIN
Blockchain Security

Smart contract security audit report

Audit Number：202106041118

Report Query Name: ins3_v2

Smart Contract Address Link：

https://github.com/ins3project/contracts

Commit Hash：

Start: aeadb6447a847f53717627dbf45d2a7bbffa6c28

Final: b6681d859ed920a345fe2f8d88654de8bff2f2f7

Start Date：2020.05.24

Completion Date：2021.06.04

Overall Result：Pass

Audit Team: Beosin (Chengdu LianAn) Technology Co. Ltd.

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	Compiler Version Security	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		SafeMath Features	Pass
		require/assert Usage	Pass
		Gas Consumption	Pass
		Visibility Specifiers	Pass
		Fallback Usage	Pass
2	General Vulnerability	Integer Overflow/Underflow	Pass
		Reentrancy	Pass
		Pseudo-random Number Generator (PRNG)	Pass
		Transaction-Ordering Dependence	Pass
		DoS (Denial of Service)	Pass
		Access Control of Owner	Pass

		Low-level Function (call/delegatecall) Security	Pass
		Returned Value Security	Pass
		tx.origin Usage	Pass
		Replay Attack	Pass
		Overriding Variables	Pass
3	Business Security	Business Logics	Pass
		Business Implementations	Pass

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 (Chengdu LianAn) 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 (Chengdu LianAn) 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 (Chengdu LianAn) 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 (Chengdu LianAn) Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin (Chengdu LianAn) Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin (Chengdu LianAn). Due to the technical limitations of any organization, this report conducted by Beosin (Chengdu LianAn) still has the possibility that the entire risk cannot be completely detected. Beosin (Chengdu LianAn) disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin (Chengdu LianAn).

Audit Results Explained:

Beosin (Chengdu LianAn) Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of smart contracts project ins3_v2, including Coding Standards, Security, and Business Logic. **The ins3_v2 project passed all audit items. The overall result is Pass. The smart contract is able to function properly.**

Coding Conventions

Check the code style that does not conform to Solidity code style.

1 Compiler Version Security

- **Description**: Check whether the code implementation of current contract contains the exposed solidity compiler bug.
- **Result**: Pass

2 Deprecated Items

- **Description**: Check whether the current contract has the deprecated items.
- **Result**: Pass

3 Redundant Code

- Description: Check whether the contract code has redundant codes.
- Result: Pass

4 SafeMath Features

- **Description**: Check whether the SafeMath has been used. Or prevents the integer overflow/underflow in mathematical operation.
- **Result**: Pass

5 require/assert Usage

- **Description**: Check the use reasonability of 'require' and 'assert' in the contract.
- **Result**: Pass

6 Gas Consumption

- **Description**: Check whether the gas consumption exceeds the block gas limitation.
- **Result**: Pass

7 Visibility Specifiers

- **Description**: Check whether the visibility conforms to design requirement.
- **Result**: Pass

8 Fallback Usage

- **Description**: Check whether the Fallback function has been used correctly in the current contract.
- **Result**: Pass

General Vulnerability

Check whether the general vulnerabilities exist in the contract.

1 Integer Overflow/Underflow

- **Description**: Check whether there is an integer overflow/underflow in the contract and the calculation result is abnormal.
- **Result**: Pass

2 Reentrancy

- **Description**: An issue when code can call back into your contract and change state, such as withdrawing ETH.

- **Result**: Pass

3 Pseudo-random Number Generator (PRNG)

- **Description**: Whether the results of random numbers can be predicted.
- **Result**: Pass

4 Transaction-Ordering Dependence

- **Description**: Whether the final state of the contract depends on the order of the transactions.
- **Result**: Pass

5 DoS (Denial of Service)

- **Description**: Whether exist DoS attack in the contract which is vulnerable because of unexpected reason.
- **Result**: Pass

6 Access Control of Owner

- **Description**:

  Whether the owner has excessive permissions, such as malicious issue, modifying the balance of others. Permission restrictions on function calls need to be restricted to avoid security problems caused by permission leakage.

  The contract permission in this contract project is controlled by the owner address, which is initially designated by the project party. This address has too much permission. It can modify the relevant parameters in the *ExchOracleMachine*, *PriceMetaInfoDB* and other contracts through the specified function to control the contract reward calculation logic. If the parameters are abnormal, it may affect the user's use and income.

  In addition, the address in the *ITFCoinHolder* contract can be used to mint the ITF by setting the specified address to call the *reward* function. If the permission is too large, once the key is lost or malicious invocation caused by other reasons will cause an abnormal ITF liquidity.

- **Suggestion**:

  It is recommended to improve the relevant permission control logic, add the voting governance mechanism to control the permission balance, and avoid malicious calls caused by the leakage of private keys and other reasons.

- **Result: After communicating with the project party, the project party stated that the business expansion in the later stage of the project requires the permission of coin minting, and the upper limit of coin minting has been set, and that it will keep the private key properly.**

7 Low-level Function (call/delegatecall) Security

- **Description**: Check whether the usage of low-level functions like call/delegatecall have vulnerabilities.
- **Result**: Pass

8 Returned Value Security

- **Description**: Check whether the function checks the return value and responds to it accordingly.

- **Result**: Pass

9 tx.origin Usage

- **Description**: Check the use secure risk of 'tx.origin' in the contract. In this project, the contract
- **Result**: Pass

10 Replay Attack

- **Description**: Check whether the implement possibility of Replay Attack exists in the contract.
- **Result**: Pass

11 Overriding Variables

- **Description**: Check whether the variables have been overridden and lead to wrong code execution.
- **Result**: Pass

Business Security

Check whether the business is secure.

The ins3_v2 project implements user participation by issuing ERC20 tokens of the corresponding product to insured users as vouchers; it implements investor underwriting by issuing ERC721 tokens to insured users as vouchers, and the oracle voting is carried out by collateralizing stable coins. In addition, the project also implements the function of collateralizing ERC721 tokens for mining ITF tokens, collateralizing LP tokens for mining ITF tokens, and the function of flash loans that use user underwriting funds as principal.

1. User insurance

- **Description**:

1) Each insurance product will correspond to an on-chain *Ins3ProductTokenV2* contract. After obtaining the insurance price and node signature from the node server provided by the project party, users can call the *buy* function to purchase this insurance product. The *Ins3ProductTokenV2* contract will call *transferFrom* to transfer user tokens to this contract, and the corresponding amount of ERC20 tokens will be minted and returned to the user as a voucher. The tokens can be transferred at will. The contract also provides a method to buy Channel through the channel business, this method the agent will directly share a certain percentage of the user's insured amount (initial 1%, owner permission can be modified), but will not affect the user's amount of insurance.

2) The user can surrender the insurance by calling the *withdraw* function of the corresponding *Ins3ProductTokenV2* contract. The surrender will destroy the ERC20 token certificate held by the user and return part of the user's principal. The specific ratio is the user's currency holdings multiplied by the current insurance unit price multiplied by the surrender percentage (Initial 70%, owner permission can be modified).

3) After the insurance expires, the owner permission account will confirm and call the *rejectPaid* function to set the product status as non-compensated, and the premium will be proportionally credited to the *StakingPoolTokenV2* contract and the project address for dividends, or call the *approvePaid* function to set the product status to claim, and the premium Enter the *StakingPool* contract to settle claims.

- **Related contract**: *Ins3ProductTokenBaseV2*、*Ins3ProductTokenV2*、*StakingPool*、*ClaimPool*、*StakingPoolTokenV2*

- **Warning**:

1) The insurance unit price is provided by the node server of the project party. If the node server is abnormal, the price will be abnormal.
2) The final result of the insurance needs to be confirmed by the owner permission. If the owner's private key is lost, it will be impossible to settle the claim normally or maliciously.

- **Result**: After communication with the project party, in order to avoid risks such as flash loan attacks and malicious manipulation of voting caused by the use of on-chain price feed, the project party chose to continue to retain the off-chain price feed and owner confirmation mechanism.

2. Investor underwriting

- **Description**:

1) Investors can underwrite through the *newTokenHolder* method of the *StakingPoolTokenV2* contract, which will allow the insurer to choose multiple insurance products for underwriting, but the amount of a single underwriting product does not exceed the actual mortgage amount, and the total amount of all insured products must not exceed the actual mortgage The amount of leverage (default 10 times), and the supported product support principal must be the same, and the expiration time must be the same. The contract will cast an ERC721 token as the user's underwriting certificate and send it to the user's account.

2) The contract allows users to withdraw the underwriting, but there must be no unfinished insurance products and insufficient remaining underwriting capacity in the underwriting products. Users who withdraw their capital before all the insured products are over will not receive underwriting rewards, and if there are already insured products that need to be settled before the withdrawal, the corresponding principal will also be deducted.

3) When all insured products normally end, the insurer will allocate premiums proportionally based on the product of the coverage amount and the power of the coverage time. If there is a claim in the insured product that affects the principal, the reward allocated will also be reduced accordingly.

- **Related contract**: *StakingPoolTokenV2*、*StakingPoolV2*

- **Suggestion**: When the principal is deducted from the claim settlement of the user's underwriting product, the total amount of pledge time for other products has not decreased, resulting in part of the premium will not be allocated to anyone, and it is recommended to update the user's holdings.

- **Result**: fixed

3 Oracle and node related

- **Description**:

Users can vote on events through *ExchOracleMachine* contract mortgage tokens, and each address will only be able to vote for mortgage once. If you have voted, the mortgaged tokens will be locked (the default is one month, the owner permission can be modified), and the user must manually collect the ITF token rewards generated by the mortgage before redeeming the mortgaged tokens, otherwise it will be lost. By default, the ITF reward is 8% of the total amount of ITF divided equally among all oracles, and is distributed in eight phases. Each time the specified premium is reached, it will enter the next phase. The above ratio can be controlled by the owner's permission.

- **Related contract**: *ExchOracleMachine*、*OracleMachine*、*OracleNode*、*OracleNodeMgr*

- **Warning**:

The final voting result of the oracle machine does not directly affect whether the product is claimed or not, and it needs to be triggered after the owner's permission is confirmed.

- **Result**: Pass

4 Underwriting token mortgage

- **Description**:

The ERC721 tokens obtained by the user after underwriting can be mortgaged in the *StakingTokenMintPool* contract to obtain ITF tokens. The annualized profit is 20% of the fixed principal value (default, owner permission can be adjusted). When all insured products expire, Earnings are cut off. The principal ratio in the value conversion is provided by the node server.

- **Related contract**: *StakingTokenMintPool*

- **Warning**:

The ITF and principal prices in value conversion are provided by the node server. If the node server fails or the private key is lost, it will affect the normal price conversion and cause losses.

- **Result**: Pass

5 LP token mortgage

- **Description**:

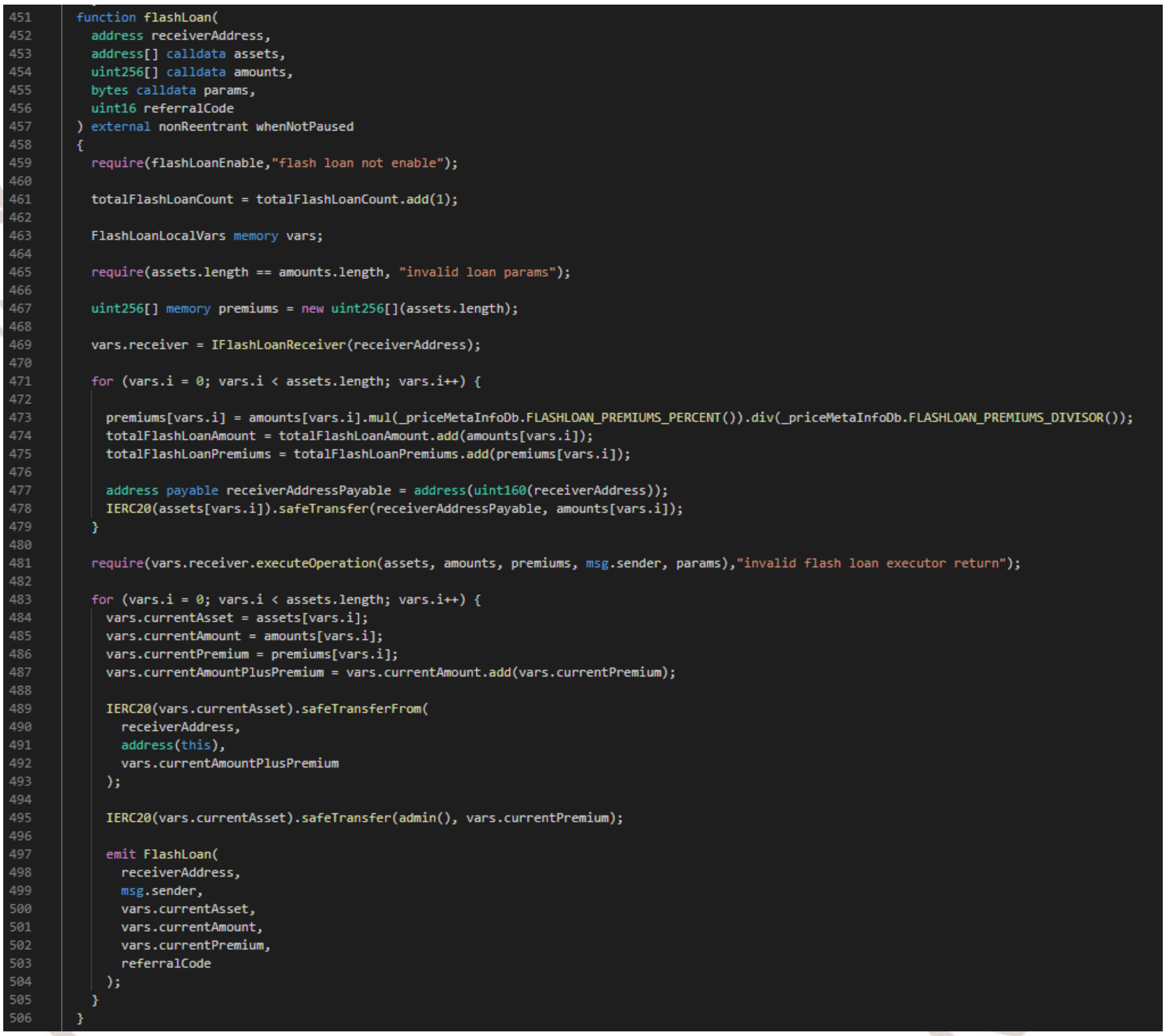
Users can mortgage the LP tokens obtained by adding liquidity to the *LiquidMintPoolMgr* contract to obtain ITF tokens, and the rewards will be distributed proportionally according to the time and amount of mortgaged LP tokens. The total reward of each mortgage pool is distributed proportionally according to its number of "nodes".

- **Related contract**: *LiquidMintPoolMgr*
- **Result**: Pass

6. Flash Loan

- **Description**:

The *StakingPoolToken* contract implements the lightning loan function, which is closed by default, and the default handling fee is 0.09% (owner permissions can be modified).



```
function flashLoan(
  address receiverAddress,
  address[] calldata assets,
  uint256[] calldata amounts,
  bytes calldata params,
  uint16 referralCode
) external nonReentrant whenNotPaused
{
  require(flashLoanEnable,"flash loan not enable");

  totalFlashLoanCount = totalFlashLoanCount.add(1);

  FlashLoanLocalVars memory vars;

  require(assets.length == amounts.length, "invalid loan params");

  uint256[] memory premiums = new uint256[](assets.length);

  vars.receiver = IFlashLoanReceiver(receiverAddress);

  for (vars.i = 0; vars.i < assets.length; vars.i++) {

    premiums[vars.i] = amounts[vars.i].mul(_priceMetaInfoDb.FLASHLOAN_PREMIUMS_PERCENT()).div(_priceMetaInfoDb.FLASHLOAN_PREMIUMS_DIVISOR());
    totalFlashLoanAmount = totalFlashLoanAmount.add(amounts[vars.i]);
    totalFlashLoanPremiums = totalFlashLoanPremiums.add(premiums[vars.i]);

    address payable receiverAddressPayable = address(uint160(receiverAddress));
    IERC20(assets[vars.i]).safeTransfer(receiverAddressPayable, amounts[vars.i]);
  }

  require(vars.receiver.executeOperation(assets, amounts, premiums, msg.sender, params),"invalid flash loan executor return");

  for (vars.i = 0; vars.i < assets.length; vars.i++) {
    vars.currentAsset = assets[vars.i];
    vars.currentAmount = amounts[vars.i];
    vars.currentPremium = premiums[vars.i];
    vars.currentAmountPlusPremium = vars.currentAmount.add(vars.currentPremium);

    IERC20(vars.currentAsset).safeTransferFrom(
      receiverAddress,
      address(this),
      vars.currentAmountPlusPremium
    );

    IERC20(vars.currentAsset).safeTransfer(admin(), vars.currentPremium);

    emit FlashLoan(
      receiverAddress,
      msg.sender,
      vars.currentAsset,
      vars.currentAmount,
      vars.currentPremium,
      referralCode
    );
  }
}
```

Figure 1 source code of functions *flashloan*

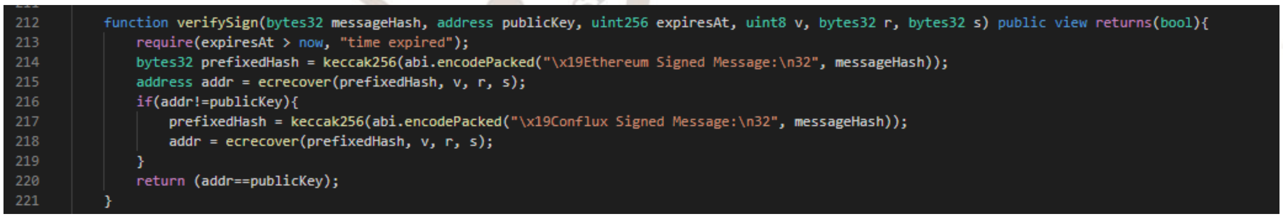
- **Related contract**: *StakingPoolTokenV2*

- **Result**: Pass

7. Other contracts

In addition to the above functions, the ins3_v2 project has also implemented the following contracts:

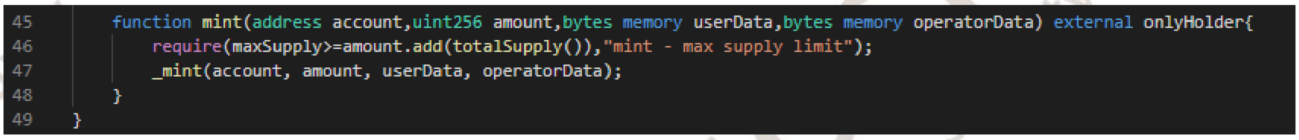
1) The *PriceMetaInfoDB* contract is used to store some of the parameters in each contract. The owner permission account can modify the parameters. The parameters include the agent's dividend ratio, lightning loan commission, project party's premium share ratio and the annualization of the underwriting certificate mortgage income, etc. The contract also implements the node signature verification function *verifySign*.



```
    function verifySign(bytes32 messageHash, address publicKey, uint256 expiresAt, uint8 v, bytes32 r, bytes32 s) public view returns(bool){
        require(expiresAt > now, "time expired");
        bytes32 prefixedHash = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", messageHash));
        address addr = ecrecover(prefixedHash, v, r, s);
        if(addr!=publicKey){
            prefixedHash = keccak256(abi.encodePacked("\x19Conflux Signed Message:\n32", messageHash));
            addr = ecrecover(prefixedHash, v, r, s);
        }
        return (addr==publicKey);
    }
```

Figure 2 source code of functions *verifySign*

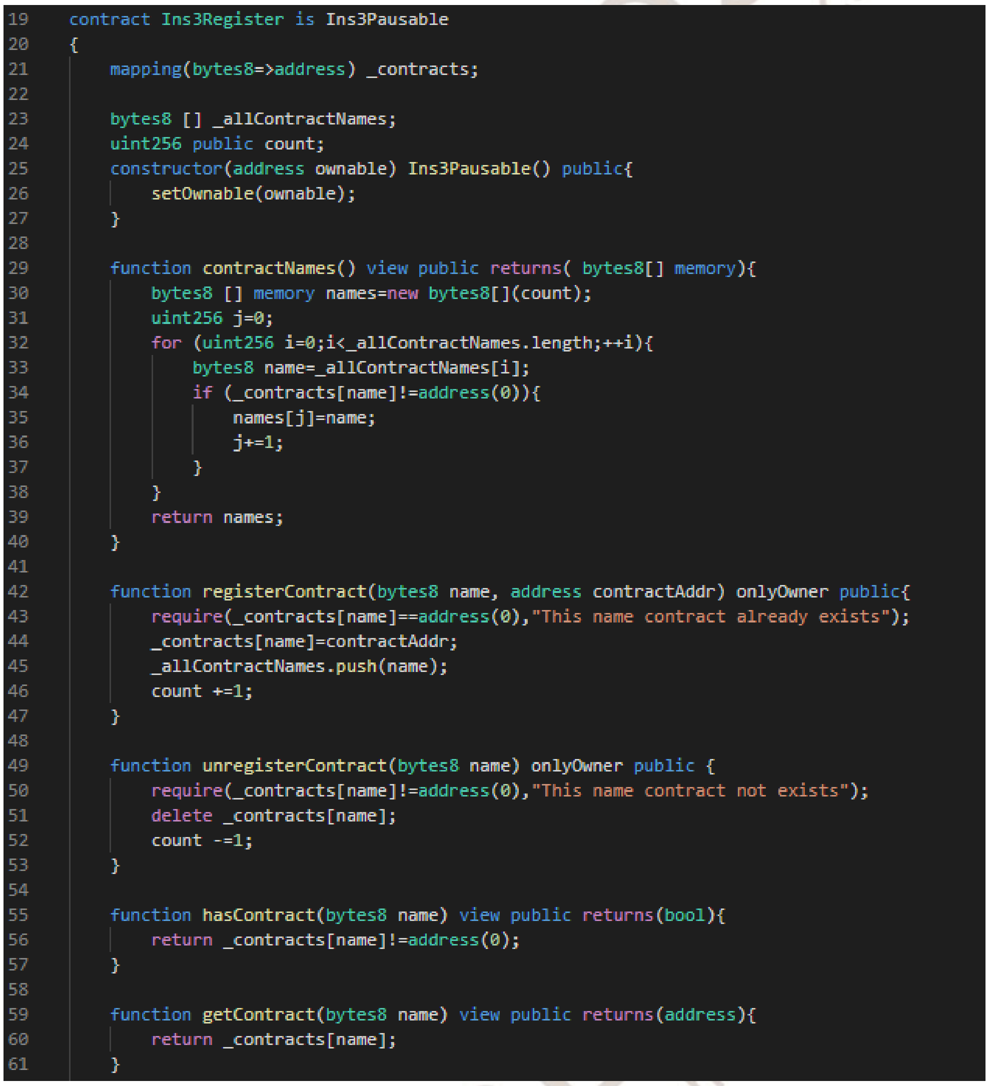
2) The *ITFCoin* contract implements an ERC777 token. The minting limit *maxSupply* will be specified during deployment. After deployment, only the address with holder permission can mint tokens.



```
    function mint(address account,uint256 amount,bytes memory userData,bytes memory operatorData) external onlyHolder{
        require(maxSupply>=amount.add(totalSupply()),"mint - max supply limit");
        _mint(account, amount, userData, operatorData);
    }
}
```

Figure 3 source code of functions *mint*

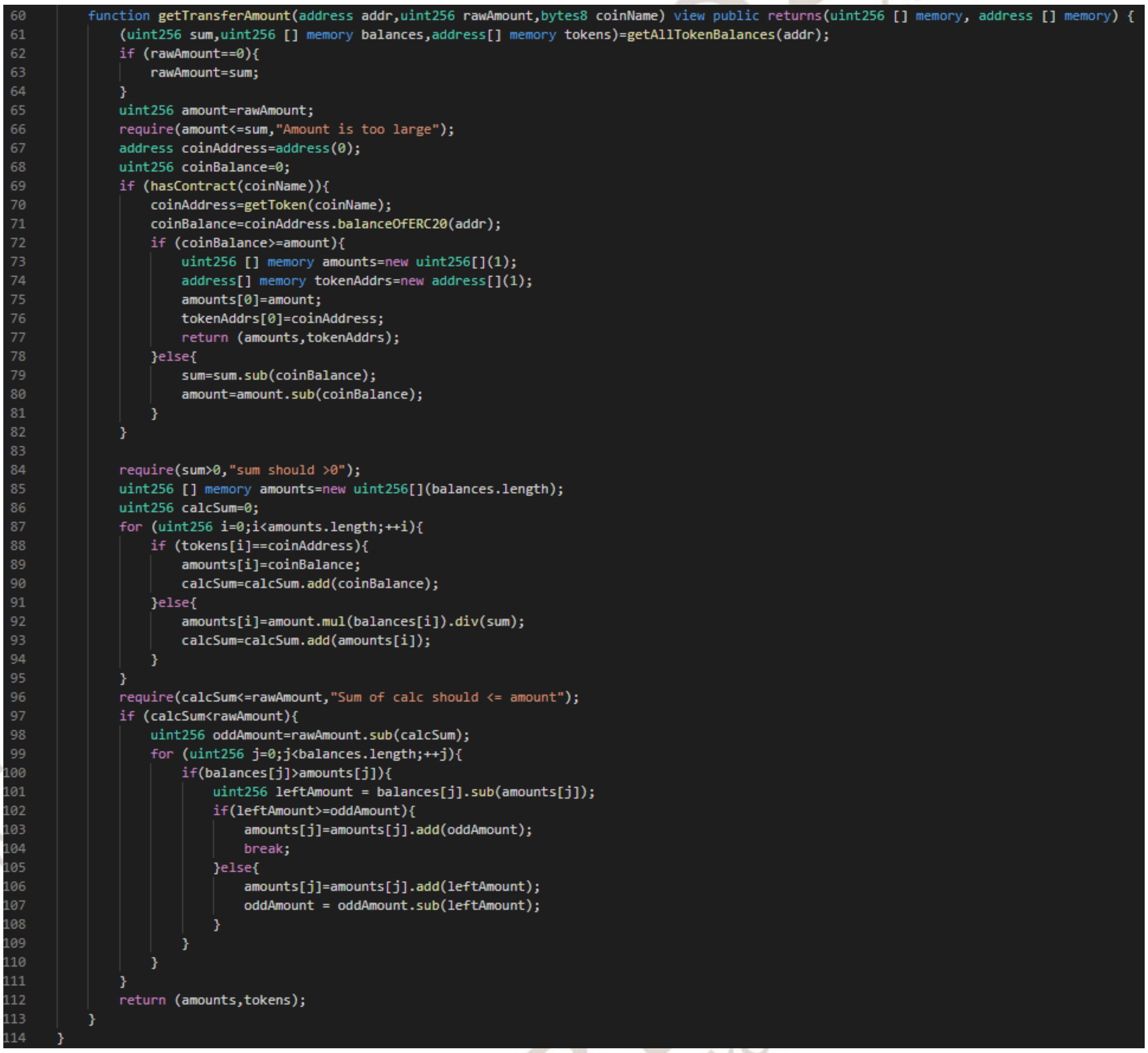
3) *Ins3Register* contract, which stores the correspondence between contract name and contract address, and is used for other contract queries. The owner permission address can add the corresponding relationship.



```
contract Ins3Register is Ins3Pausable
{
    mapping(bytes8=>address) _contracts;

    bytes8 [] _allContractNames;
    uint256 public count;
    constructor(address ownable) Ins3Pausable() public{
        setOwnable(ownable);
    }

    function contractNames() view public returns( bytes8[] memory){
        bytes8 [] memory names=new bytes8[](count);
        uint256 j=0;
        for (uint256 i=0;i<_allContractNames.length;++i){
            bytes8 name=_allContractNames[i];
            if (_contracts[name]!=address(0)){
                names[j]=name;
                j+=1;
            }
        }
        return names;
    }

    function registerContract(bytes8 name, address contractAddr) onlyOwner public{
        require(_contracts[name]==address(0),"This name contract already exists");
        _contracts[name]=contractAddr;
        _allContractNames.push(name);
        count +=1;
    }

    function unregisterContract(bytes8 name) onlyOwner public {
        require(_contracts[name]!=address(0),"This name contract not exists");
        delete _contracts[name];
        count -=1;
    }

    function hasContract(bytes8 name) view public returns(bool){
        return _contracts[name]!=address(0);
    }

    function getContract(bytes8 name) view public returns(address){
        return _contracts[name];
    }
```

Figure 4 source code of Ins3Register contract

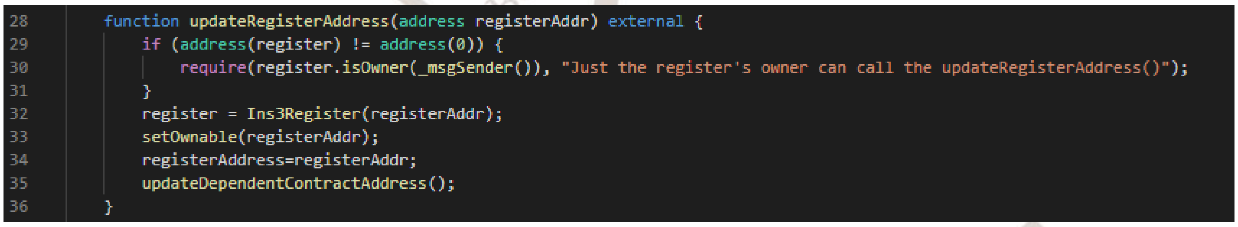
4) The *ERC20TokenRegister* contract stores the corresponding relationship between the token name and the token address, and implements a variety of token transaction schemes. The *getTransferAmount* function can return the corresponding payment plan according to the balance of multiple tokens at the specified address. This feature has been deprecated in the V2 version of underwriting and pledge.



```
    function getTransferAmount(address addr,uint256 rawAmount,bytes8 coinName) view public returns(uint256 [] memory, address [] memory) {
        (uint256 sum,uint256 [] memory balances,address[] memory tokens)=getAllTokenBalances(addr);
        if (rawAmount==0){
            rawAmount=sum;
        }
        uint256 amount=rawAmount;
        require(amount<=sum,"Amount is too large");
        address coinAddress=address(0);
        uint256 coinBalance=0;
        if (hasContract(coinName)){
            coinAddress=getToken(coinName);
            coinBalance=coinAddress.balanceOfERC20(addr);
            if (coinBalance>=amount){
                uint256 [] memory amounts=new uint256[](1);
                address[] memory tokenAddrs=new address[](1);
                amounts[0]=amount;
                tokenAddrs[0]=coinAddress;
                return (amounts,tokenAddrs);
            }else{
                sum=sum.sub(coinBalance);
                amount=amount.sub(coinBalance);
            }
        }

        require(sum>0,"sum should >0");
        uint256 [] memory amounts=new uint256[](balances.length);
        uint256 calcSum=0;
        for (uint256 i=0;i<amounts.length;++i){
            if (tokens[i]==coinAddress){
                amounts[i]=coinBalance;
                calcSum=calcSum.add(coinBalance);
            }else{
                amounts[i]=amount.mul(balances[i]).div(sum);
                calcSum=calcSum.add(amounts[i]);
            }
        }
        require(calcSum<=rawAmount,"Sum of calc should <= amount");
        if (calcSum<rawAmount){
            uint256 oddAmount=rawAmount.sub(calcSum);
            for (uint256 j=0;j<balances.length;++j){
                if(balances[j]>amounts[j]){
                    uint256 leftAmount = balances[j].sub(amounts[j]);
                    if(leftAmount>=oddAmount){
                        amounts[j]=amounts[j].add(oddAmount);
                        break;
                    }else{
                        amounts[j]=amounts[j].add(leftAmount);
                        oddAmount = oddAmount.sub(leftAmount);
                    }
                }
            }
        }
        return (amounts,tokens);
    }
}
```

Figure 5 source code of functions *getTransferAmount*

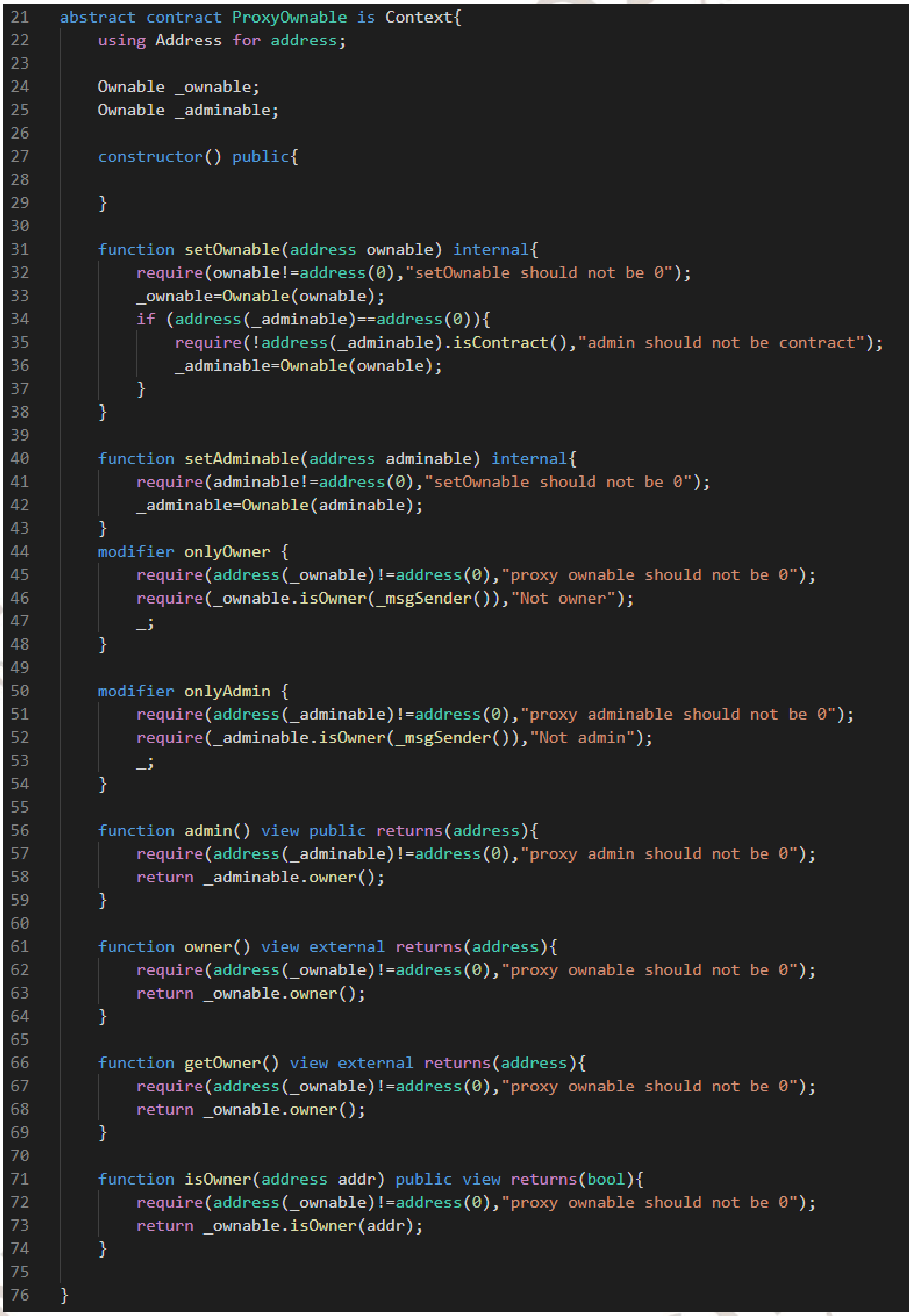
5) The *IUpgradable* contract implements the upgrade function of the contract. By inheriting this contract, the owner permission of the successor will point to the owner address of the *Ins3Register* contract, and the contract address read in the successor will be updated through the *updateRegisterAddress* function.



```
    function updateRegisterAddress(address registerAddr) external {
        if (address(register) != address(0)) {
            require(register.isOwner(_msgSender()), "Just the register's owner can call the updateRegisterAddress()");
        }
        register = Ins3Register(registerAddr);
        setOwnable(registerAddr);
        registerAddress=registerAddr;
        updateDependentContractAddress();
    }
```

Figure 6 source code of functions *updateRegisterAddress*

6) The *ProxyOwnable* contract implements the owner proxy function. By inheriting this contract and calling the setOwnable function, the owner permission value can be contracted to the owner of the specified contract.



```solidity
abstract contract ProxyOwnable is Context{
    using Address for address;

    Ownable _ownable;
    Ownable _adminable;

    constructor() public{

    }

    function setOwnable(address ownable) internal{
        require(ownable!=address(0),"setOwnable should not be 0");
        _ownable=Ownable(ownable);
        if (address(_adminable)==address(0)){
            require(!address(_adminable).isContract(),"admin should not be contract");
            _adminable=Ownable(ownable);
        }
    }

    function setAdminable(address adminable) internal{
        require(adminable!=address(0),"setOwnable should not be 0");
        _adminable=Ownable(adminable);
    }
    modifier onlyOwner {
        require(address(_ownable)!=address(0),"proxy ownable should not be 0");
        require(_ownable.isOwner(_msgSender()),"Not owner");
        _;
    }

    modifier onlyAdmin {
        require(address(_adminable)!=address(0),"proxy adminable should not be 0");
        require(_adminable.isOwner(_msgSender()),"Not admin");
        _;
    }

    function admin() view public returns(address){
        require(address(_adminable)!=address(0),"proxy admin should not be 0");
        return _adminable.owner();
    }

    function owner() view external returns(address){
        require(address(_ownable)!=address(0),"proxy ownable should not be 0");
        return _ownable.owner();
    }

    function getOwner() view external returns(address){
        require(address(_ownable)!=address(0),"proxy ownable should not be 0");
        return _ownable.owner();
    }

    function isOwner(address addr) public view returns(bool){
        require(address(_ownable)!=address(0),"proxy ownable should not be 0");
        return _ownable.isOwner(addr);
    }

}
```

Figure 7 source code of contract ProxyOwnable

- **Related contract**: *ProxyOwnable*、*IUpgradable*、*ERC20TokenRegister*、*Ins3Register*、*ITFCoin*、*PriceMetaInfoDB*

- **Result**: Pass

8. New features and changes in V2

1) The underwriting changes from supporting multiple tokens to supporting only one token, and it is mandatory that all products of a policy support the same tokens and have the same expiration time.

2) The newly added *ClaimPool* contract is used for claim settlement of DeFi products. After the user has been insured, the user needs to pledge the voucher token generated by the DeFi product to ClaimPool. When the mortgage reaches a certain percentage, it is judged that the claim can be settled. Insured users will get claims, and insured users will get DeFi certificates.

3) The newly added *NFTErc20Adapter* contract is used to convert the underwriting NFT into an equivalent ERC20, and users can use this ERC20 to pledge mining to earn income.

Conclusion

Beosin (Chengdu LianAn) conducted a detailed audit on the design and code implementation of the ins3_v2 project smart contract. The problems discovered by the audit team during the audit process have been notified to the project party to repair. The biggest risk point of the project comes from the project party’s private key management. Because the owner permission and the service node have high permission, if the private key is lost, it will be very large. loss. The overall audit result of the smart contract of the ins3_v2 project is **pass**.

BEOSIN

Blockchain Security

Official Website

https://lianantech.com

E-mail

vaas@lianantech.com

Twitter

https://twitter.com/Beosin_com