Degree of secrecy ★ | Project team open

Smart Contract Audit Report

Security status

Safe

★ ★ ★ ★ ★

Principal tester: Knownsec blockchain security team

Version Summary

Content	Date	Version
Editing Document	2021/08/05	V1.0

Report Information

Title	Version	Document Number	Type
decredit Smart Contract Audit Report	V1.0	1e13103e306840cd96e7a410cb708ee2	Open to project team

Copyright Notice

Table of Contents

1. Introduction

The effective test time of this report is from From **August 3, 2021** to **August 5, 2021** . During this period, the reputation setting of **decredit Smart Contract Code** and the security and standardization of borrowed and redeemed assets will be audited and used as the statistical basis for the report.

The scope of this smart contract security audit does not include external contract calls, new attack methods that may appear in the future, and code after contract upgrades or tampering. (With the development of the project, the smart contract may add a new pool, New functional modules, new external contract calls, etc.), does not include front-end security and server security.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 3). **The smart contract code of the decredit** is comprehensively assessed as **SAFE**.

Results of this smart contract security audit: SAFE

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

Report information of this audit:

Report Number: 1e13103e306840cd96e7a410cb708ee2

Report query address link:

https://attest.im/attestation/searchResult?qurey=1e13103e306840cd96e7a410cb708ee2

Target information of the decredit audit:

Target information	
Project name	decredit

Token address	Unitroller 0x3b1dD467c50fF62E1061aD832649d62B9A946209 DCConfig 0xfb9f8C31985792b8F64a8e906DB43b208440f329 CreditOracle 0x651a5dD386A766Bc7BCCd2e14Aedaef550123537 ChainlinkAdaptor 0x87C95A430620c822dFF9f1d3676559Ddd49Da255 CompoundLens 0xAF69eDe7f9855dc0a2F07935D7B97Aa83E5F0fD4 Maximillion 0x623D304a81CB23Df02C0f33a0fa46F25259f625c dBNB 0x5Ac50C5A1612CE7f7B78B0a369c2701e17C9af17 dUSDT 0xBF586363eA34AE1A7C4F84BC5542692aE0Fe77dA USDT 0x55d398326f99059fF775485246999027B3197955 dETH 0x370B505d4AE4398AdC75a7413Ace6c1B6Dc8c65a ETH 0x2170Ed0880ac9A755fd29B2688956BD959F933F8 dBTCB 0x100D51466D725D2B3320b6076c632e143BCF31C4 BTCB 0x7130d2A12B9BCbFAe4f2634d864A1Ee1Ce3Ead9c

	dBUSD 0x81c2819FB01ffF17e2fF03Aa8a881B859befa682 BUSD 0xe9e7CEA3DedcA5984780Bafc599bD69ADd087D56
Code type	BSC smart contract code
Code language	Solidity

Contract documents and hash:

Contract documents	MD5
CreditOracle.sol	f47601cf2ecda0b7881a2f96c743d361
IPriceCollector.sol	ccfcfb1b35b4419f80346f4eb2788fa5
Ownable.sol	78b9f82d2958d8a3de2b76614c1c1478
TetherToken.sol	24433b0e801b7a5f66d7f8d9b96eaf4e
MockPriceOracle.sol	238c6698a5de4e2dbb3be690e6622da1
DCSimplePriceOracle.sol	c623b50eaa390cd70371f6db0f75f57a
HFILToken.sol	cfd5aea051c9aff3006e088929be9051
ChainlinkAggregatorV3Interface.sol	c6c6d716432f2d48fd8f578a7508b049
DCConfig.sol	49099ea5e22cb76a4e940daf404204d0
EIP20NonStandardInterface.sol	233b54ba1f055b8c2b9ea1c1dd3608f3
ComptrollerInterface.sol	6cdb79a0889d8c37c8c5642d0813b37a

Comptroller.sol	883749072372182ee0d6bd906a409198
CErc20.sol	28b1fd837b212644a245dcc8f6e7d0a4
Exponential.sol	eaf3b583cd84e37d7a28223c88d8a114
PriceOracle.sol	3a863051264cbd4ef4f328bbf8bc5650
CToken.sol	391d90a2d338738d4d54cfb0f31d372d
CErc20Delegator.sol	6c470bc9603affc0cc6c7be0d517a63a
CErc20Delegate.sol	f07e57da8e44d23d2d99737d27126e1d
CDaiDelegate.sol	b0cc054b06bb0680192c6ba2692402a9
PriceOracleProxy.sol	45c08b15fcc3288c1b72423868dd9fbd
SafeMath.sol	51e3bce6f64c8cbdb9f6a6c4d8ccaa1b
ErrorReporter.sol	4360952ef4d39b9c916546d941dca6d1
Unitroller.sol	481805e9c40a023d03e6d832a9ffd667
Migrations.sol	ca8d6ca8a6edf34f149a5095a8b074c9
CarefulMath.sol	b1f19e9f4ff15ac6673cd0d4ce242b01
JumpRateModel.sol	1f0d388f2ccdbda7b0926a6bc70db8e7
Timelock.sol	32f77c167fae6e68ce3869f94a20775f
EIP20Interface.sol	fa93e469fb558e63a43784a83f62fc89
InterestRateModel.sol	f8e83b5b683c7150fb53ea27df826cbe
ComptrollerStorage.sol	c325b1d599b7d0d359b274c68b3ff0f2
CompoundLens.sol	29536534aa532fe995a8a1f864022222

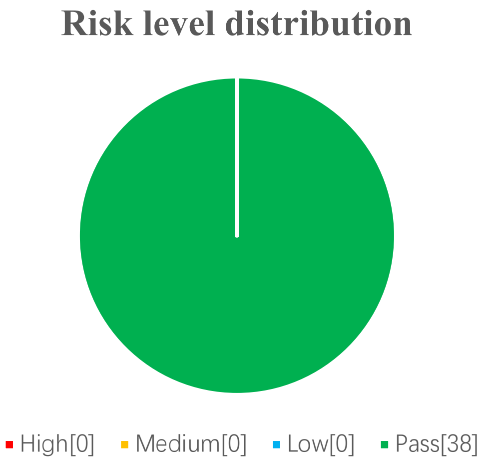
WhitePaperInterestRateModel.sol	4336abae2e23b69bb562af199d944b6e
Reservoir.sol	113b5858eb4b59e5cd76bd651d9524c5
GovernorAlpha.sol	aa6329672b8f078e017ff40cf4735501
Comp.sol	707f728a3eb2bcd67f37ed5ed3c67244
DAIInterestRateModelV2.sol	74d4598de0d5d16aa22a6b9965492236
CErc20Immutable.sol	f935ea442c903467e41ef569ba2cd4f0
CEther.sol	d2c11be85e4647509551a00d7f54f104
Maximillion.sol	ff32fc4d8bb119d9c6e8945d95774658
SimplePriceOracle.sol	2c2c0b93265e2988d7450ceae6a5c8fd
CTokenInterfaces.sol	c5c81b131f5b7bbf75d0c6f7128e8aaf
ChainlinkAdaptor.sol	3e8127d4e6c4c99e0240f59f91db8eb2
DCPriceOracle.sol	16cad23c91af70510560354c6ed047c3
DefaultBSCInterestModel.sol	f9ebaf2a43084dbe8ebccd9d37b70eda
SToken.sol	7a5525be9adaf02f024d58a8539b0f4e
DCtroller.sol	ae69ab60d50654dc10cf9136cd63629c
BSCJumpInterestModel.sol	00f367390b1147733b972c74dfa91a50

2. Code vulnerability analysis

2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level:

Vulnerability risk level statistics table			
High	Medium	Low	Pass
0	0	0	38



2.2 Audit Result

Result of audit			
Audit Target	**Audit**	**Status**	**Audit Description**
Business security testing	**ChainlinkAdaptor contract obtains external price function**	Pass	After testing, there is no such safety vulnerability.
	CreditOracle contract setting reputation level function	Pass	After testing, there is no such safety vulnerability.
	Comptroller contract entry and exit function	Pass	After testing, there is no such safety vulnerability.
	DCtroller contract query user remaining liquidity function	Pass	After testing, there is no such safety vulnerability.
	CToken contract initialization function	Pass	After testing, there is no such safety vulnerability.
	CToken contract update interest rate function	Pass	After testing, there is no such safety vulnerability.
	CToken contract minting function	Pass	After testing, there is no such safety vulnerability.
	CToken contract redemption token function	Pass	After testing, there is no such safety vulnerability.
	CToken contract lending function	Pass	After testing, there is no such safety vulnerability.
	CToken contract liquidation loan function	Pass	After testing, there is no such safety vulnerability.

	SToken contract seize function function	Pass	After testing, there is no such safety vulnerability.
Basic code vulnerability detection	**Compiler version security**	Pass	After testing, there is no such safety vulnerability.
	Redundant code	Pass	After testing, there is no such safety vulnerability.
	Use of safe arithmetic library	Pass	After testing, there is no such safety vulnerability.
	Not recommended encoding	Pass	After testing, there is no such safety vulnerability.
	Reasonable use of require/assert	Pass	After testing, there is no such safety vulnerability.
	fallback function safety	Pass	After testing, there is no such safety vulnerability.
	tx.origin authentication	Pass	After testing, there is no such safety vulnerability.
	Owner permission control	Pass	After testing, there is no such safety vulnerability.
	Gas consumption detection	Pass	After testing, there is no such safety vulnerability.
	call injection attack	Pass	After testing, there is no such safety vulnerability.
	Low-level function safety	Pass	After testing, there is no such safety vulnerability.
	Vulnerability of additional token issuance	Pass	After testing, there is no such safety vulnerability.

	Access control defect detection	Pass	After testing, there is no such safety vulnerability.
	Numerical overflow detection	Pass	After testing, there is no such safety vulnerability.
	Arithmetic accuracy error	Pass	After testing, there is no such safety vulnerability.
	Wrong use of random number detection	Pass	After testing, there is no such safety vulnerability.
	Unsafe interface use	Pass	After testing, there is no such safety vulnerability.
	Variable coverage	Pass	After testing, there is no such safety vulnerability.
	Uninitialized storage pointer	Pass	After testing, there is no such safety vulnerability.
	Return value call verification	Pass	After testing, there is no such safety vulnerability.
	Transaction order dependency detection	Pass	After testing, there is no such safety vulnerability.
	Timestamp dependent attack	Pass	After testing, there is no such safety vulnerability.
	Denial of service attack detection	Pass	After testing, there is no such safety vulnerability.
	Fake recharge vulnerability detection	Pass	After testing, there is no such safety vulnerability.
	Reentry attack detection	Pass	After testing, there is no such safety vulnerability.
	Replay attack detection	Pass	After testing, there is no such safety vulnerability.

	Rearrangement attack detection	Pass	After testing, there is no such safety vulnerability.

3. Analysis of code audit results

3.1. ChainlinkAdaptor contract obtains external price function【PASS】

Audit analysis: The getUnderlyingPrice function of the ChainlinkAdaptor.sol contract realizes the function of the project to obtain the asset price under the specified contract from the chainlink oracle interface.

```
function getUnderlyingPrice(CToken cToken) external view returns (uint) { //knownsec// Query the current price under the ctoken contract            if (SToken(address(cToken)).isNativeToken()) {
            return 1e18;
        }

        address asset = address(CErc20(address(cToken)).underlying());//knownsec//   Get the address of the asset token under the corresponding ctoken
        ChainlinkAggregatorV3Interface priceSource = assetsPriceSources[asset]; //knownsec// Get the external interface address corresponding to the asset token in the asset mapping table
        if (address(priceSource) == address(0x0)) {
            return getUnderlyingPriceFromFallback(cToken);
        }

        uint256 priceFromChainlink = getUnderlyingPriceFromChainlink(priceSource, cToken); //knownsec//   Return price from external contract
        if (priceFromChainlink == 0) {
            return getUnderlyingPriceFromFallback(cToken); //knownsec// Get the price at the backup oracle interface
        }
        return priceFromChainlink;
    }

    function getUnderlyingPriceFromFallback(CToken cToken) public view returns (uint) {
        if (address(fallbackPriceOracle) != address(0x0)) {
```

```
        return fallbackPriceOracle.getUnderlyingPrice(cToken); //knownsec// Try to get the price from the backup price oracle machine
    }
    return 0;
}

function getUnderlyingPriceFromChainlink(ChainlinkAggregatorV3Interface chainlinkPriceSource, CToken cToken) view internal returns(uint256) { //knownsec// Get external asset token prices from chainlink
    uint256 assetPriceDecimals = chainlinkPriceSource.decimals();//knownsec// Obtaining source data price accuracy
    address asset = address(CErc20(address(cToken)).underlying());//knownsec// Get the token address specified under the corresponding ctoken

    uint256 assetDecimals = CErc20(address(asset)).decimals();//knownsec// Obtain asset price accuracy
    uint256 assetPriceInUsd = getPrice(chainlinkPriceSource);

    uint256 nativeTokenPriceDecimals = nativeTokenPriceSource.decimals();
    uint256 nativeTokenPriceInUsd = getPrice(nativeTokenPriceSource);

    if (assetPriceInUsd == 0 || nativeTokenPriceInUsd == 0) {
        return 0;
    }
    if (assetPriceDecimals == nativeTokenPriceDecimals) {
        return assetPriceInUsd.mul(10 ** 18).mul(10 ** 18).div(nativeTokenPriceInUsd.mul(10 ** assetDecimals)); //knownsec// Returns the token price calculated from the oracle
    } else {//knownsec// If the token accuracy is different
        return assetPriceInUsd.mul(10 ** 18).mul(10 ** nativeTokenPriceDecimals).mul(10 ** 18).div(nativeTokenPriceInUsd.mul(10 ** assetDecimals).mul(10 ** assetPriceDecimals));
    }
```

```
    }
```

Recommendation： nothing.

3.2. CreditOracle contract setting reputation level function 【PASS】

Audit analysis: The CreditOracle.sol contract implements the use of a credit rating system to judge the amount of collateral required for a user's borrowing.

```
constructor() public {
        creditAdmin[msg.sender] = true; //knownsec//  The default deployer is a reputation administrator
        emit CreditAdminChanged(msg.sender, true);
    }
function updateCreditCollateralRatio(address account, uint creditScore) public onlyCreditAdmin {//knownsec// Update account credit rating
        require(account != address(0), "Set credit score to zero address");
        require(creditScore > 0 && creditScore <= 100, "Invalid credit score");//knownsec// Reputation points are between 0-100

        uint oldAccountCreditRatio = creditCollateralRatio[account];
        uint accountCreditRatio = calCreditCollateralRatio(creditScore); //knownsec// Calculate mortgage interest rate based on credit score
        creditCollateralRatio[account] = accountCreditRatio; //knownsec// Update the mortgage rate of the account

        emit CreditCollateralRatioChanged(account, oldAccountCreditRatio, accountCreditRatio, creditScore); //knownsec//   Trigger update event
    }
    function setCreditValueDefinition(uint _scopeFrom, uint _scopeTo, uint _creditValue) public onlyOwner { //knownsec//    // Set reputation level, each level has a corresponding range
        require(_creditValue > 0 && _creditValue <= 100, "Zero credit value");
        require(_scopeFrom > 0 && _scopeTo <= 100, "Invalid credit scope value");
```

```
        require(inScope(_scopeFrom), "Invalid credit scope value");

        creditValue[_scopeFrom] = _creditValue; //knownsec// Set the reputation score of the corresponding range
        emit CreditValueDefinitionChanged(_scopeFrom, _scopeTo, _creditValue); //knownsec// Trigger to modify the definition event of reputation score
```

Recommendation: nothing.

3.3. Comptroller contract entry and exit function 【PASS】

Audit analysis: The enterMarkets and exitMarket functions of Comptroller.sol realize the functions of users entering and exiting the designated token market. When entering a certain token market, they can perform lending and liquidation functions in that market.

```
function enterMarkets(address[] memory cTokens) public returns (uint[] memory) { //knownsec// enter the market
        uint len = cTokens.length; //knownsec// Get the length of the token market

        uint[] memory results = new uint[](len);
        for (uint i = 0; i < len; i++) {
            CToken cToken = CToken(cTokens[i]);

            results[i] = uint(addToMarketInternal(cToken, msg.sender)); //knownsec// Call internal functions to add user information to the token market
        }

        return results;
    }

    /**
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
     * @param cToken The market to enter
```

```
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */
    function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
        Market storage marketToJoin = markets[address(cToken)];

        if (!marketToJoin.isListed) { //knownsec// The added token must be in the market list published by the administrator
            // market is not listed, cannot join
            return Error.MARKET_NOT_LISTED;
        }

        if (marketToJoin.accountMembership[borrower] == true) { //knownsec// The token market users have not added
            // already joined
            return Error.NO_ERROR;
        }

        // survived the gauntlet, add to list
        // NOTE: we store these somewhat redundantly as a significant optimization
        //  this avoids having to iterate through the list for the most common use cases
        //  that is, only when we need to perform liquidity checks
        //  and not whenever we want to check if an account is in a particular market
        marketToJoin.accountMembership[borrower] = true;
        accountAssets[borrower].push(cToken);

        emit MarketEntered(cToken, borrower); //knownsec// Trigger market entry event

        return Error.NO_ERROR;
    }

    /**
     * @notice Removes asset from sender's account liquidity calculation
```

```
    * @dev Sender must not have an outstanding borrow balance in the asset,
    *   or be providing necessary collateral for an outstanding borrow.
    * @param cTokenAddress The address of the asset to be removed
    * @return Whether or not the account successfully exited the market
    */
  function exitMarket(address cTokenAddress) external returns (uint) { //knownsec// qiut the market
      CToken cToken = CToken(cTokenAddress);
      /* Get sender tokensHeld and amountOwed underlying from the cToken */
      (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.getAccountSnapshot(msg.sender); //knownsec// Obtain the caller's possession and owed amount according to the token address
      require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

      /* Fail if the sender has a borrow balance */
      if (amountOwed != 0) { //knownsec// The caller cannot have arrears
          return fail(Error.NONZERO_BORROW_BALANCE, FailureInfo.EXIT_MARKET_BALANCE_OWED);
      }

      /* Fail if the sender is not permitted to redeem all of their tokens */
      uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld); //knownsec// Determine whether the tokens can be redeemed
      if (allowed != 0) {
          return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
      }

      Market storage marketToExit = markets[address(cToken)];

      /* Return true if the sender is not already 'in' the market */
      if (!marketToExit.accountMembership[msg.sender]) { //knownsec// Determine whether the caller enters the token market
```

```
            return uint(Error.NO_ERROR);
        }

        /* Set cToken account membership to false */
        delete marketToExit.accountMembership[msg.sender]; //knownsec// Set to false, no longer a member of the token

        /* Delete cToken from the account’s list of assets */
        // load into memory for faster iteration
        CToken[] memory userAssetList = accountAssets[msg.sender]; //knownsec// Update user asset list
        uint len = userAssetList.length;
        uint assetIndex = len;
        for (uint i = 0; i < len; i++) {
            if (userAssetList[i] == cToken) {
                assetIndex = i;
                break;
            }
        }

        // We *must* have found the asset in the list or our redundant data structure is broken
        assert(assetIndex < len);

        // copy last item in list to location of item to be removed, reduce length by 1
        CToken[] storage storedList = accountAssets[msg.sender]; //knownsec// Add the last digit to the deleted position, the total length is -1
        storedList[assetIndex] = storedList[storedList.length - 1];
        storedList.length--;

        emit MarketExited(cToken, msg.sender); //knownsec// Trigger withdrawal from the token market event

        return uint(Error.NO_ERROR);
    }
```

Recommendation: nothing.

3.4. DCtroller contract query user remaining liquidity function【PASS】

Audit analysis: The getHypotheticalAccountLiquidityInternal function of DCtroller.sol inherits the Comptroller contract and realizes the change of the user's liquidity after the user's funds are lent or borrowed.

```
function getHypotheticalAccountLiquidityInternal(
        address account, //knownsec// User address
        CToken cTokenModify, //knownsec// Token market address
        uint redeemTokens,
        uint borrowAmount) internal view returns (Error, uint, uint) {

        AccountLiquidityLocalVars memory vars; // Holds all our calculation results
        uint oErr;
        MathError mErr;

        // For each asset the account is in
        CToken[] memory assets = accountAssets[account];
        for (uint i = 0; i < assets.length; i++) {
            CToken asset = assets[i]; //knownsec// Get the address of each asset

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) =
asset.getAccountSnapshot(account); //knownsec// Get the corresponding asset balance and
exchange rate mantissa
            if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant
between upgrades
                return (Error.SNAPSHOT_ERROR, 0, 0);
            }
            vars.collateralFactor = Exp({mantissa:
```

```
markets[address(asset)].collateralFactorMantissa});
            vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa}); //knownsec// Calculate the exchange rate

            // Get the normalized price of the asset
            vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset); //knownsec// Get the standard price of the asset
            if (vars.oraclePriceMantissa == 0) {
                return (Error.PRICE_ERROR, 0, 0);
            }
            vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});

            // Pre-compute a conversion factor from tokens -> ether (normalized price value)
            (mErr, vars.tokensToDenom) = mulExp3(vars.collateralFactor, vars.exchangeRate, vars.oraclePrice); //knownsec// Calculation conversion factor
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // sumCollateral += tokensToDenom * cTokenBalance
            (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(vars.tokensToDenom, vars.cTokenBalance, vars.sumCollateral); //knownsec// Calculate ctoken into the total amount of collateral, and accumulate each token
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // sumBorrowPlusEffects += oraclePrice * borrowBalance
            (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, vars.borrowBalance, vars.sumBorrowPlusEffects); //knownsec// Cumulative bid price * the sum of the borrowing quantity
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
```

```
            }

            // Calculate effects of interacting with cTokenModify
            if (asset == cTokenModify) {
                // redeem effect
                // sumBorrowPlusEffects += tokensToDenom * redeemTokens
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.tokensToDenom, redeemTokens, vars.sumBorrowPlusEffects); //knownsec// Cumulative redemption value
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }

                // borrow effect
                // sumBorrowPlusEffects += oraclePrice * borrowAmount
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice, borrowAmount, vars.sumBorrowPlusEffects); //knownsec// Calculate loan value
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }
            }
        }

        uint creditCollateralRatio = creditOracle.getCreditCollateralRatio(account);
        if (creditCollateralRatio > 0) { //knownsec// Credit Mortgage Rate
            (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(Exp({mantissa: creditCollateralRatio}), vars.sumCollateral, vars.sumCollateral); //knownsec// Calculate the amount of collateral value
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }
        }
```

```
        // These are safe, as the underflow condition is checked first
        if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
            return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
        } else {
            return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral); //knownsec// Returns the sum of loan value-the amount of collateral value
        }
    }
```

Recommendation: nothing.

3.5. CToken contract initialization function【PASS】

Audit analysis: The initialize function of CToken.sol implements the initialization of the contract, including basic information such as the token name, management contract address, interest rate model address, and initial exchange rate.

```
 function initialize(ComptrollerInterface comptroller_,
                     InterestRateModel interestRateModel_,
                     uint initialExchangeRateMantissa_,
                     string memory name_,
                     string memory symbol_,
                     uint8 decimals_) public { //knownsec// Management contract address, interest rate model address, initial exchange rate, name, abbreviation, precision
        require(msg.sender == admin, "only admin may initialize the market"); //knownsec// The caller must be admin
        require(accrualBlockNumber == 0 && borrowIndex == 0, "market may only be initialized once"); //knownsec// The token market can only be initialized once

        // Set initial exchange rate
        initialExchangeRateMantissa = initialExchangeRateMantissa_;
        require(initialExchangeRateMantissa > 0, "initial exchange rate must be greater than zero."); //knownsec// The exchange rate is required to be greater than 0
```

```
        // Set the comptroller
        uint err = _setComptroller(comptroller_); //knownsec// Set the address of the auditor, and request to return as no error
        require(err == uint(Error.NO_ERROR), "setting comptroller failed");

        // Initialize block number and borrow index (block number mocks depend on comptroller being set)
        accrualBlockNumber = getBlockNumber(); //knownsec// Get the current block number
        borrowIndex = mantissaOne;

        // Set the interest rate model (depends on block number / borrow index)
        err = _setInterestRateModelFresh(interestRateModel_); //knownsec// Setting up an interest rate model
        require(err == uint(Error.NO_ERROR), "setting interest rate model failed");

        name = name_;
        symbol = symbol_;
        decimals = decimals_;

        // The counter starts true to prevent changing it from zero to non-zero (i.e. smaller cost/refund)
        _notEntered = true; //knownsec// Initially true
    }
```

Recommendation: nothing.

3.6. CToken contract update interest rate function 【PASS】

Audit analysis: The accrueInterest function implements the function of updating the contract interest rate and user loan information.

```
function accrueInterest() public returns (uint) { //knownsec// Apply the accrued interest to the total borrowing and reserve amount
        /* Remember the initial block number */
```

```
uint currentBlockNumber = getBlockNumber(); //knownsec// Current block number
uint accrualBlockNumberPrior = accrualBlockNumber; //knownsec// Last updated block number

/* Short-circuit accumulating 0 interest */
if (accrualBlockNumberPrior == currentBlockNumber) {
    return uint(Error.NO_ERROR);
}

/* Read the previous values out of storage */
uint cashPrior = getCashPrior(); //knownsec// Store value before reading
uint borrowsPrior = totalBorrows; //knownsec//  Total outstanding loans
uint reservesPrior = totalReserves; //knownsec// Total reserve
uint borrowIndexPrior = borrowIndex;

/* Calculate the current borrow interest rate */
uint borrowRateMantissa = interestRateModel.getBorrowRate(cashPrior, borrowsPrior, reservesPrior); //knownsec// Calculate the current borrowing rate
require(borrowRateMantissa <= borrowRateMaxMantissa, "borrow rate is absurdly high");

/* Calculate the number of blocks elapsed since the last accrual */
(MathError mathErr, uint blockDelta) = subUInt(currentBlockNumber, accrualBlockNumberPrior); //knownsec// Calculate the block number passed
require(mathErr == MathError.NO_ERROR, "could not calculate block delta");

/*
 * Calculate the interest accumulated into borrows and reserves and the new index:
 *  simpleInterestFactor = borrowRate * blockDelta
 *  interestAccumulated = simpleInterestFactor * totalBorrows
 *  totalBorrowsNew = interestAccumulated + totalBorrows
 *  totalReservesNew = interestAccumulated * reserveFactor + totalReserves
 *  borrowIndexNew = simpleInterestFactor * borrowIndex + borrowIndex
```

```
    */

    Exp memory simpleInterestFactor;
    uint interestAccumulated;
    uint totalBorrowsNew;
    uint totalReservesNew;
    uint borrowIndexNew;

    (mathErr, simpleInterestFactor) = mulScalar(Exp({mantissa: borrowRateMantissa}), blockDelta); //knownsec// Lending rate*blocks passed
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED, uint(mathErr));
    }

    (mathErr, interestAccumulated) = mulScalarTruncate(simpleInterestFactor, borrowsPrior); //knownsec// Interest = lending interest rate * blocks passed * total borrowing amount
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED, uint(mathErr));
    }

    (mathErr, totalBorrowsNew) = addUInt(interestAccumulated, borrowsPrior); //knownsec// New repayable amount = interest + loaned principal
    if (mathErr != MathError.NO_ERROR) {
        return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED, uint(mathErr));
    }
```

```
        (mathErr, totalReservesNew) = mulScalarTruncateAddUInt(Exp({mantissa: reserveFactorMantissa}), interestAccumulated, reservesPrior); //knownsec// New total reserves = interest * reserve factor + total reserves
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED, uint(mathErr));
        }

        (mathErr, borrowIndexNew) = mulScalarTruncateAddUInt(simpleInterestFactor, borrowIndexPrior, borrowIndexPrior); //knownsec// Update the accumulative variable of return
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED, uint(mathErr));
        }

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

        /* We write the previously calculated values into storage */
        accrualBlockNumber = currentBlockNumber;
        borrowIndex = borrowIndexNew;
        totalBorrows = totalBorrowsNew;
        totalReserves = totalReservesNew;

        /* We emit an AccrueInterest event */
        emit AccrueInterest(cashPrior, interestAccumulated, borrowIndexNew, totalBorrowsNew); //knownsec// Trigger update event

        return uint(Error.NO_ERROR);
    }
```

Recommendation: nothing.

3.7. CToken contract minting function【PASS】

Audit analysis: The mintFresh function realizes the function that users can obtain cToken by transferring designated tokens.

```
function mintFresh(address minter, uint mintAmount) internal returns (uint, uint) {
        /* Fail if mint not allowed */
        uint allowed = comptroller.mintAllowed(address(this), minter, mintAmount); //knownsec// Check if you can mint the address, return 0 to pass
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.MINT_COMPTROLLER_REJECTION, allowed), 0);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) { //knownsec// Ensure that the block numbers are equal
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.MINT_FRESHNESS_CHECK), 0);
        }

        MintLocalVars memory vars;

        (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal(); //knownsec// Get conversion factor
        if (vars.mathErr != MathError.NO_ERROR) {
            return (failOpaque(Error.MATH_ERROR, FailureInfo.MINT_EXCHANGE_RATE_READ_FAILED, uint(vars.mathErr)), 0);
        }

        ///////////////////////
```

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

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

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

        (vars.mathErr, vars.mintTokens) = divScalarByExpTruncate(vars.actualMintAmount, Exp({mantissa: vars.exchangeRateMantissa})); //knownsec// Calculate the number of coins based on the current market rate
        require(vars.mathErr == MathError.NO_ERROR, "MINT_EXCHANGE_CALCULATION_FAILED");

        /*
         * We calculate the new total supply of cTokens and minter token balance, checking for overflow:
         *  totalSupplyNew = totalSupply + mintTokens
         *  accountTokensNew = accountTokens[minter] + mintTokens
         */
        (vars.mathErr, vars.totalSupplyNew) = addUInt(totalSupply, vars.mintTokens); //knownsec// Update the total supply of cToken
```

```
        require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED");

        (vars.mathErr, vars.accountTokensNew) = addUInt(accountTokens[minter], vars.mintTokens); //knownsec// Update minter token cToken balance
        require(vars.mathErr == MathError.NO_ERROR, "MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED");

        /* We write previously calculated values into storage */
        totalSupply = vars.totalSupplyNew;
        accountTokens[minter] = vars.accountTokensNew;

        /* We emit a Mint event, and a Transfer event */
        emit Mint(minter, vars.actualMintAmount, vars.mintTokens); //knownsec// Trigger minting and transfer events
        emit Transfer(address(this), minter, vars.mintTokens);

        /* We call the defense hook */
        comptroller.mintVerify(address(this), minter, vars.actualMintAmount, vars.mintTokens); //knownsec// Mint verification function
        return (uint(Error.NO_ERROR), vars.actualMintAmount);
    }
```

Recommendation: nothing.

3.8. CToken contract redemption token function【PASS】

Audit analysis: The redeemFresh function enables users to redeem their mortgage assets by selling their cToken.

```
function redeemFresh(address payable redeemer, uint redeemTokensIn, uint redeemAmountIn) internal returns (uint) {
        require(redeemTokensIn == 0 || redeemAmountIn == 0, "one of redeemTokensIn or redeemAmountIn must be zero");
```

```
        RedeemLocalVars memory vars;

        /* exchangeRate = invoke Exchange Rate Stored() */
        (vars.mathErr, vars.exchangeRateMantissa) = exchangeRateStoredInternal(); //knownsec// Recall storage rate
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_RATE_READ_FAILED, uint(vars.mathErr));
        }

        /* If redeemTokensIn > 0: */
        if (redeemTokensIn > 0) {
            /*
             * We calculate the exchange rate and the amount of underlying to be redeemed:
             *  redeemTokens = redeemTokensIn
             *  redeemAmount = redeemTokensIn x exchangeRateCurrent
             */
            vars.redeemTokens = redeemTokensIn;

            (vars.mathErr, vars.redeemAmount) = mulScalarTruncate(Exp({mantissa: vars.exchangeRateMantissa}), redeemTokensIn); //knownsec// Number of redemptions = accuracy * number of tokens
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED, uint(vars.mathErr));
            }
        } else {
            /*
             * We get the current exchange rate and calculate the amount to be redeemed:
             *  redeemTokens = redeemAmountIn / exchangeRate
             *  redeemAmount = redeemAmountIn
             */
```

```
            (vars.mathErr, vars.redeemTokens) = divScalarByExpTruncate(redeemAmountIn, Exp({mantissa: vars.exchangeRateMantissa})); //knownsec// Number of tokens = number/precision
            if (vars.mathErr != MathError.NO_ERROR) {
                return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED, uint(vars.mathErr));
            }

            vars.redeemAmount = redeemAmountIn;
        }

        /* Fail if redeem not allowed */
        uint allowed = comptroller.redeemAllowed(address(this), redeemer, vars.redeemTokens); //knownsec// Check whether the transfer is allowed
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.REDEEM_COMPTROLLER_REJECTION, allowed);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return fail(Error.MARKET_NOT_FRESH, FailureInfo.REDEEM_FRESHNESS_CHECK);
        }

        /*
         * We calculate the new total supply and redeemer balance, checking for underflow:
         *  totalSupplyNew = totalSupply - redeemTokens
         *  accountTokensNew = accountTokens[redeemer] - redeemTokens
         */
        (vars.mathErr, vars.totalSupplyNew) = subUInt(totalSupply, vars.redeemTokens); //knownsec// Update the total circulation of ctoken
        if (vars.mathErr != MathError.NO_ERROR) {
```

```
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED, uint(vars.mathErr));
        }

        (vars.mathErr, vars.accountTokensNew) = subUInt(accountTokens[redeemer], vars.redeemTokens); //knownsec// Update user balance
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR, FailureInfo.REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

        /* Fail gracefully if protocol has insufficient cash */
        if (getCashPrior() < vars.redeemAmount) {
            return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.REDEEM_TRANSFER_OUT_NOT_POSSIBLE); //knownsec// Determine whether the number of tokens available in the contract is sufficient to pay
        }

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

        /*
         * We invoke doTransferOut for the redeemer and the redeemAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken has redeemAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects occurred.
         */
        doTransferOut(redeemer, vars.redeemAmount); //knownsec// Transfer money to the redeemer
```

```
        /* We write previously calculated values into storage */
        totalSupply = vars.totalSupplyNew;
        accountTokens[redeemer] = vars.accountTokensNew;

        /* We emit a Transfer event, and a Redeem event */
        emit Transfer(redeemer, address(this), vars.redeemTokens); //knownsec// Trigger transfer event
        emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens); //knownsec// Trigger a redemption event

        /* We call the defense hook */
        comptroller.redeemVerify(address(this), redeemer, vars.redeemAmount, vars.redeemTokens);

        return uint(Error.NO_ERROR);
    }
```

Recommendation: nothing.

3.9. CToken contract lending function【PASS】

Audit analysis: The borrowFresh function realizes the function that users can borrow from the contract by mortgage their own assets as a guarantee.

```
function borrowFresh(address payable borrower, uint borrowAmount) internal returns (uint) { //knownsec// Borrowing address, loan amount
        /* Fail if borrow not allowed */
        uint allowed = comptroller.borrowAllowed(address(this), borrower, borrowAmount); //knownsec// Determine if it can be lent
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.BORROW_COMPTROLLER_REJECTION, allowed);
        }
```

```
/* Verify market's block number equals current block number */
if (accrualBlockNumber != getBlockNumber()) { //knownsec// Verify block number
    return fail(Error.MARKET_NOT_FRESH, FailureInfo.BORROW_FRESHNESS_CHECK);
}

/* Fail gracefully if protocol has insufficient underlying cash */
if (getCashPrior() < borrowAmount) { //knownsec// The token balance in the contract is greater than the borrowed amount
    return fail(Error.TOKEN_INSUFFICIENT_CASH, FailureInfo.BORROW_CASH_NOT_AVAILABLE);
}

BorrowLocalVars memory vars;

/*
 * We calculate the new borrower and total borrow balances, failing on overflow:
 *  accountBorrowsNew = accountBorrows + borrowAmount //knownsec// Update user loan amount
 *  totalBorrowsNew = totalBorrows + borrowAmount
 */
(vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(borrower); //knownsec// Get the amount of user borrowing
if (vars.mathErr != MathError.NO_ERROR) {
    return failOpaque(Error.MATH_ERROR, FailureInfo.BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
}

(vars.mathErr, vars.accountBorrowsNew) = addUInt(vars.accountBorrows, borrowAmount);
if (vars.mathErr != MathError.NO_ERROR) {
    return failOpaque(Error.MATH_ERROR,
```

```
FailureInfo.BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED,
uint(vars.mathErr));
        }

        (vars.mathErr, vars.totalBorrowsNew) = addUInt(totalBorrows, borrowAmount);
//knownsec// Update total loan amount
        if (vars.mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED, uint(vars.mathErr));
        }

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

        /*
         * We invoke doTransferOut for the borrower and the borrowAmount.
         *  Note: The cToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the cToken borrowAmount less of cash.
         *  doTransferOut reverts if anything goes wrong, since we can't be sure if side effects
occurred.
         */
        doTransferOut(borrower, borrowAmount); //knownsec// Transfer to the borrower

        /* We write the previously calculated values into storage */
        accountBorrows[borrower].principal = vars.accountBorrowsNew; //knownsec// Update
the total amount borrowed by the borrower
        accountBorrows[borrower].interestIndex = borrowIndex; //knownsec// Update
borrowing rate
        totalBorrows = vars.totalBorrowsNew; //knownsec//  Update total loan amount

        /* We emit a Borrow event */
        emit Borrow(borrower, borrowAmount, vars.accountBorrowsNew,
```

```
vars.totalBorrowsNew); //knownsec// Trigger loan event

        /* We call the defense hook */
        comptroller.borrowVerify(address(this), borrower, borrowAmount);

        return uint(Error.NO_ERROR);
    }
```

Recommendation: nothing.

3.10. CToken contract liquidation loan function【PASS】

Audit analysis: The createPair function of the contract will determine whether the transaction pair exists, and if it does not exist, it will add a liquid mining transaction pair. The function code is standardized, and no obvious security problems have been found.

```
function liquidateBorrowFresh(address liquidator, address borrower, uint repayAmount, CTokenInterface cTokenCollateral) internal returns (uint, uint) {
        /* Fail if liquidate not allowed */
        uint allowed = comptroller.liquidateBorrowAllowed(address(this), address(cTokenCollateral), liquidator, borrower, repayAmount); //knownsec// Check if liquidation is allowed
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.LIQUIDATE_COMPTROLLER_REJECTION, allowed), 0);
        }

        /* Verify market's block number equals current block number */
        if (accrualBlockNumber != getBlockNumber()) {
            return (fail(Error.MARKET_NOT_FRESH, FailureInfo.LIQUIDATE_FRESHNESS_CHECK), 0);
        }
```

```
/* Verify cTokenCollateral market's block number equals current block number */
if (cTokenCollateral.accrualBlockNumber() != getBlockNumber()) {
    return (fail(Error.MARKET_NOT_FRESH, FailureInfo.LIQUIDATE_COLLATERAL_FRESHNESS_CHECK), 0);
}

/* Fail if borrower = liquidator */
if (borrower == liquidator) { //knownsec// Liquidators cannot be borrowers
    return (fail(Error.INVALID_ACCOUNT_PAIR, FailureInfo.LIQUIDATE_LIQUIDATOR_IS_BORROWER), 0);
}

/* Fail if repayAmount = 0 */
if (repayAmount == 0) {
    return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_ZERO), 0);
}

/* Fail if repayAmount = -1 */
if (repayAmount == uint(-1)) {
    return (fail(Error.INVALID_CLOSE_AMOUNT_REQUESTED, FailureInfo.LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX), 0);
}


/* Fail if repayBorrow fails */
(uint repayBorrowError, uint actualRepayAmount) = repayBorrowFresh(liquidator, borrower, repayAmount); //knownsec// Liquidator repays the loan
if (repayBorrowError != uint(Error.NO_ERROR)) {
    return (fail(Error(repayBorrowError), FailureInfo.LIQUIDATE_REPAY_BORROW_FRESH_FAILED), 0);
}
```

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

/* We calculate the number of collateral tokens that will be seized */
(uint amountSeizeError, uint seizeTokens) = comptroller.liquidateCalculateSeizeTokens(address(this), address(cTokenCollateral), actualRepayAmount); //knownsec// Calculate the number of tokens worth of collateral
require(amountSeizeError == uint(Error.NO_ERROR), "LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED");

/* Revert if borrower collateral token balance < seizeTokens */
require(cTokenCollateral.balanceOf(borrower) >= seizeTokens, "LIQUIDATE_SEIZE_TOO_MUCH"); //knownsec// Borrower's token balance> the amount of tokens exchanged for collateral

// If this is also the collateral, run seizeInternal to avoid re-entrancy, otherwise make an external call
uint seizeError;
if (address(cTokenCollateral) == address(this)) { //knownsec// If the collateral is the contract token
    seizeError = seizeInternal(address(this), liquidator, borrower, seizeTokens); //knownsec// Call internal function for confiscation
} else {
    seizeError = cTokenCollateral.seize(liquidator, borrower, seizeTokens); //knownsec// The liquidator seizes the borrower's seizeTokens amount of tokens
}

/* Revert if seize tokens fails (since we cannot be sure of side effects) */
require(seizeError == uint(Error.NO_ERROR), "token seizure failed");

/* We emit a LiquidateBorrow event */
emit LiquidateBorrow(liquidator, borrower, actualRepayAmount,
```

```
address(cTokenCollateral), seizeTokens); //knownsec// Trigger liquidation event

        /* We call the defense hook */
        comptroller.liquidateBorrowVerify(address(this), address(cTokenCollateral), liquidator,
borrower, actualRepayAmount, seizeTokens); //knownsec// Verify liquidation event
        return (uint(Error.NO_ERROR), actualRepayAmount);
    }
```

Recommendation: nothing.

3.11. SToken contract seize function function【PASS】

Audit analysis: The seize function of SToken.sol inherits the CToken contract. The seize function is an external attribute, and the seizeInternal function can be called internally to update the token balance. The seizeInternal function realizes the function that the liquidator obtains the liquidation reward paid by the borrower by repaying the borrower.

```
function seizeInternal(address seizerToken, address liquidator, address borrower, uint seizeTokens)
internal returns (uint) {
        /* Fail if seize not allowed */
        uint allowed = comptroller.seizeAllowed(address(this), seizerToken, liquidator, borrower,
seizeTokens); //knownsec // Check if liquidation is allowed, return 0 if passed
        if (allowed != 0) {
            return failOpaque(Error.COMPTROLLER_REJECTION,
FailureInfo.LIQUIDATE_SEIZE_COMPTROLLER_REJECTION, allowed);
        }

        /* Fail if borrower = liquidator */
        if (borrower == liquidator) { //knownsec// Judging that the liquidator cannot be a
borrower
            return fail(Error.INVALID_ACCOUNT_PAIR,
FailureInfo.LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER);
        }
```

```
        MathError mathErr;
        uint borrowerTokensNew;
        uint liquidatorTokensNew;
        uint safetyVaultTokensNew;
        uint safetyVaultTokens;
        uint liquidatorSeizeTokens;

        (liquidatorSeizeTokens, safetyVaultTokens) =
DCtroller(address(comptroller)).dcConfig().calculateSeizeTokenAllocation(seizeTokens,
DCtroller(address(comptroller)).liquidationIncentiveMantissa());
        address safetyVault = DCtroller(address(comptroller)).dcConfig().safetyVault();
        /*
         * We calculate the new borrower and liquidator token balances, failing on
underflow/overflow:
         *  borrowerTokensNew = accountTokens[borrower] - seizeTokens
         *  liquidatorTokensNew = accountTokens[liquidator] + seizeTokens
         */
        (mathErr, borrowerTokensNew) = subUInt(accountTokens[borrower], seizeTokens);
//knownsec// Update the borrower's ctoken token balance
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED, uint(mathErr));
        }

        (mathErr, liquidatorTokensNew) = addUInt(accountTokens[liquidator],
liquidatorSeizeTokens); //knownsec// Increase the liquidator's ctoken token balance
        if (mathErr != MathError.NO_ERROR) {
            return failOpaque(Error.MATH_ERROR,
FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));
        }

        (mathErr, safetyVaultTokensNew) = addUInt(accountTokens[safetyVault],
```

```
safetyVaultTokens);
        if (mathErr != MathError.NO_ERROR) {
            return                                  failOpaque(Error.MATH_ERROR,
FailureInfo.LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED, uint(mathErr));
        }

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

        /* We write the previously calculated values into storage */
        accountTokens[borrower] = borrowerTokensNew; //knownsec// Update token balance
        accountTokens[liquidator] = liquidatorTokensNew;
        accountTokens[safetyVault] = safetyVaultTokensNew;

        /* Emit a Transfer event */
        emit Transfer(borrower, liquidator, liquidatorSeizeTokens); //knownsec// Trigger ctoken
transfer event
        emit Transfer(borrower, safetyVault, safetyVaultTokens);

        /* We call the defense hook */
        comptroller.seizeVerify(address(this), seizerToken, liquidator, borrower, seizeTokens);
//knownsec// Verification of the confiscation incident has no practical significance

        return uint(Error.NO_ERROR);
    }
```

Recommendation： nothing.

4. Basic code vulnerability detection

4.1. Compiler version security【PASS】

Check whether a safe compiler version is used in the contract code implementation.

Audit result: After testing, the smart contract code has formulated the compiler version 0.6.12 within the major version, and there is no such security problem.

Recommendation: nothing.

4.2. Redundant code【PASS】

Check whether the contract code implementation contains redundant code.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.3. Use of safe arithmetic library【PASS】

Check whether the SafeMath safe arithmetic library is used in the contract code implementation.

Audit result: After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

Recommendation: nothing.

4.4. Not recommended encoding【PASS】

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.5. Reasonable use of require/assert【PASS】

Check the rationality of the use of require and assert statements in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.6. Fallback function safety【PASS】

Check whether the fallback function is used correctly in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.7. tx.origin authentication【PASS】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract makes the contract vulnerable to attacks like phishing.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.8. Owner permission control【PASS】

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.9. Gas consumption detection【PASS】

Check whether the consumption of gas exceeds the maximum block limit.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.10. call injection attack 【PASS】

When the call function is called, strict permission control should be done, or the function called by the call should be written dead.

Audit result: After testing, the smart contract does not use the call function, and this vulnerability does not exist.

Recommendation: nothing.

4.11. Low-level function safety 【PASS】

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.12. Vulnerability of additional token issuance 【PASS】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.13. Access control defect detection 【PASS】

Different functions in the contract should set reasonable permissions.

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation： nothing.

4.14. Numerical overflow detection 【PASS】

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.15. Arithmetic accuracy error【PASS】

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages-Solidity does not float Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same level of calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.16. Incorrect use of random numbers【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually more public than they appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.17. Unsafe interface usage【PASS】

Check whether unsafe interfaces are used in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.18. Variable coverage【PASS】

Check whether there are security issues caused by variable coverage in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.19. Uninitialized storage pointer【PASS】

In solidity, a special data structure is allowed to be a struct structure, and the local variables in the function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

Audit result: After testing, the smart contract code does not use structure, and there is no such problem.

Recommendation： nothing.

4.20. Return value call verification【PASS】

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

In Solidity, there are transfer(), send(), call.value() and other currency transfer methods, which can all be used to send BNB to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when call.value fails to be sent; all available gas will be passed for calling (can be Limit by passing in gas_value parameters), which cannot effectively prevent reentry attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to BNB sending failure.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.21. Transaction order dependency【PASS】

Since miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.22. Timestamp dependency attack【PASS】

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it only needs to verify whether the timestamp is later than the previous block and The error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that depend on the timestamp in the contract code implementation.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.23. Denial of service attack【PASS】

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.24. Fake recharge vulnerability【PASS】

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

Audit result: After testing, the security problem does not exist in the smart contract code.

Recommendation: nothing.

4.25. Reentry attack detection【PASS】

Re-entry vulnerability is the most famous Ethereum smart contract vulnerability, which once led to the fork of Ethereum (The DAO hack).

The call.value() function in Solidity consumes all the gas it receives when it is used to send BNB. When the call.value() function to send BNB occurs before the actual reduction of the sender’s account balance, There is a risk of reentry attacks.

Audit results: After auditing, the vulnerability does not exist in the smart contract code.

Recommendation: nothing.

4.26. Replay attack detection【PASS】

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts.

Audit results: After testing, the smart contract does not use the call function, and this vulnerability does not exist.

Recommendation: nothing.

4.27. Rearrangement attack detection【PASS】

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping (mapping), so that the attacker has the opportunity to store their own information in the contract in.

Audit results: After auditing, the vulnerability does not exist in the smart contract code.

Recommendation: nothing.

5. Appendix A: Vulnerability rating standard

Smart contract vulnerability rating standards	
Level	**Level Description**
High	Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose BNB or tokens. Access loopholes, etc.; Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.; Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending BNB to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.
Medium	High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc.
Low	Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of BNB or tokens to trigger, vulnerabilities where attackers cannot directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk.

6. Appendix B: Introduction to auditing tools

6.1. Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

6.2. Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

6.3. securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues at any time.

6.4. Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

6.5. MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm the error.

6.6. ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

6.7. ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

6.8. Remix-ide

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

6.9. **Knownsec Penetration Tester Special Toolkit**

Pen-Tester tools collection is created by KnownSec team. It contains plenty of Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

Beijing KnownSec Information Technology Co., Ltd.

Advisory telephone +86(10)400 060 9587
E-mail sec@knownsec.com
Website www.knownsec.com
Address wangjing soho T2-B2509,Chaoyang District, Beijing