

Smart contract security audit report



Cashcow smart contract security audit report

Audit Team : Noneage security team

Audit Date : June 22 , 2021

Cashcow Smart Contract Security Audit Report

1. Overview

On June 19, 2021, the security team of Noneage Technology received the security audit request of the **Cashcow project**. The team will conduct a report on the **Cashcow smart contract** from June 19, 2021 to June 22, 2021. During the audit process, the security audit experts of Noneage Technology communicate with the relevant interface people of the Cashcow project, maintain information symmetry, conduct security audits under controllable operational risks, and try to avoid project generation and operation during the test process. Cause risks.

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

Audit Report MD5: 07894980EC338649D9209AAF9D038D55

2. Background

2.1 Project Description

Project name: Cashcow

official website: <https://bsc.cashcow.finance>

Contract type: DeFi Token contract

Code language: Solidity

Official GitHub repository address: <https://github.com/cashcowfinance>

Contract documents: Cow.sol, Chef.sol, Comptroller.sol, JumpRateModelV2.sol

2.2 Audit Range

Cashcow officially provides the contract address on the chain:

Cow

<https://bscscan.com/address/0xF823f18d13DF1ffdCed206708D389Dd455Bb802b>

Chef

<https://bscscan.com/address/0xbfcaB1627c4fB86A055DE4B8a56D46e625F51C0B>

Comptroller

<https://bscscan.com/address/0xFD97A1bd5F655237c1559F3fecb8E4b995bff3c1>

JumpRateModelV2

<https://bscscan.com/address/0x3E841c2a040859Fbe9743C5885bBc9015F63aFA0>

3. Contract Structure Analysis

3.1 Directory Structure

```
|
|
└─Cashcow
|   Chef.sol
|   Cow.sol
|   Comptroller.sol
|   JumpRateModelV2.sol
```

3.2 Cashcow contract

Contract

Cow

- allowance(address account, address spender)
- approve(address spender, uint rawAmount)
- balanceOf(address account)
- transfer(address dst, uint rawAmount)
- transferFrom(address src, address dst, uint rawAmount)
- delegate(address delegatee)
- delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32 r, bytes32 s)
- getCurrentVotes(address account)
- getPriorVotes(address account, uint blockNumber)
- _delegate(address delegator, address delegatee)
- _transferTokens(address src, address dst, uint96 amount)
- _moveDelegates(address srcRep, address dstRep, uint96 amount)
- _writeCheckpoint(address delegatee, uint32 nCheckpoints, uint96 oldVotes, uint96 newVotes)
- safe32(uint n, string memory errorMessage)
- safe96(uint n, string memory errorMessage)
- add96(uint96 a, uint96 b, string memory errorMessage)
- sub96(uint96 a, uint96 b, string memory errorMessage)
- getChainId()

Chef

- `poolLength()`
- `updateCowPerBlock(uint256 _cowPerBlock)`
- `add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate)`
- `set(uint256 _pid, uint256 _allocPoint, bool _withUpdate)`
- `getMultiplier(uint256 _from, uint256 _to)`
- `pendingCow(uint256 _pid, address _user)`
- `setReferral(address _referral)`
- `massUpdatePools()`
- `updatePool(uint256 _pid)`
- `deposit(uint256 _pid, uint256 _amount)`
- `withdraw(uint256 _pid, uint256 _amount)`
- `emergencyWithdraw(uint256 _pid)`
- `safeCowTransfer(address _to, uint256 _amount)`
- `grantCowInternal(address _to, uint _amount)`
- `_grantCow(address recipient, uint amount)`



Comptroller

- `payReferral(address addr, uint256 amount)`
- `getAssetsIn(address account)`
- `checkMembership(address account, CToken cToken)`
- `enterMarkets(address[] memory cTokens)`
- `addToMarketInternal(CToken cToken, address borrower)`
- `exitMarket(address cTokenAddress)`
- `mintAllowed(address cToken, address minter, uint mintAmount)`
- `mintVerify(address cToken, address minter, uint actualMintAmount, uint mintTokens)`
- `redeemAllowed(address cToken, address redeemer, uint redeemTokens)`
- `redeemAllowedInternal(address cToken, address redeemer, uint redeemTokens)`
- `redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens)`
- `borrowAllowed(address cToken, address borrower, uint borrowAmount)`
- `borrowVerify(address cToken, address borrower, uint borrowAmount)`
- `transferAllowed(address cToken, address src, address dst, uint transferTokens)`
- `transferVerify(address cToken, address src, address dst, uint transferTokens)`
- `getAccountLiquidity(address account)`
- `getAccountLiquidityInternal(address account)`
- `liquidateCalculateSeizeTokens(address cTokenBorrowed, address cTokenCollateral, uint actualRepayAmount)`
- `_setPriceOracle(PriceOracle newOracle)`
- `_setCloseFactor(uint newCloseFactorMantissa)`
- `setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa)`
- `setLiquidationIncentive(uint newLiquidationIncentiveMantissa)`
- `supportMarket(CToken cToken)`
- `addMarketInternal(address cToken)`
- `setPauseGuardian(address newPauseGuardian)`
- `setMintPaused(CToken cToken, bool state)`
- `setBorrowPaused(CToken cToken, bool state)`
- `setTransferPaused(bool state)`
- `setSeizePaused(bool state)`
- `become(Unitroller unitroller)`
- `setCreditLimit(address protocol, uint creditLimit)`
- `adminOrInitializing()`
- `refreshCowSpeeds()`
- `refreshCowSpeedsInternal()`
- `updateCowSupplyIndex(address cToken)`

- `updateCowBorrowIndex(address cToken, Exp memory marketBorrowIndex)`
- `distributeSupplierCow(address cToken, address supplier, bool distributeAll)`
- `distributeBorrowerCow(address cToken, address borrower, Exp memory marketBorrowIndex, bool distributeAll)`
- `transferCow(address user, uint userAccrued, uint threshold)`
- `updateContributorRewards(address contributor)`
- `claimCow(address holder)`
- `claimCow(address holder, CToken[] memory cTokens)`
- `claimCow(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppliers)`
- `grantCowInternal(address user, uint amount)`
- `setReferral(address referral)`
- *`grantCow(address recipient, uint amount)`*
- *`setContributorCowSpeed(address contributor, uint cowSpeed)`*
- *`setCowRate(uint cowRate)`*
- `_addCowMarkets(address[] memory cTokens)`
- `_addCowMarketInternal(address cToken)`
- `_dropCowMarket(address cToken)`
- `getAllMarkets()`
- `getBlockNumber()`
- `getCowAddress()`

BaseJumpRateModelV2

- `updateJumpRateModel(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_)`
- `utilizationRate(uint cash, uint borrows, uint reserves)`
- `getBorrowRateInternal(uint cash, uint borrows, uint reserves)`
- `getSupplyRate(uint cash, uint borrows, uint reserves, uint reserveFactorMantissa)`
- `updateJumpRateModelInternal(uint baseRatePerYear, uint multiplierPerYear, uint jumpMultiplierPerYear, uint kink_)`

JumpRateModelV2

- `getBorrowRate(uint cash, uint borrows, uint reserves)`

4. Audit details

4.1 Risk distribution

Name	Risk level	Repair status
Integer Overflow	No	normal
Variable update	No	normal
Administrator authority issues	No	normal
Numerical accuracy	No	normal
Default visibility	No	normal
tx.origin authentication	No	normal
Wrong constructor	No	normal
Unverified return value	No	normal
Insecure random number	No	normal
Timestamp dependent	No	normal
Transaction order dependence	No	normal
Delegatecall	No	normal
Call	No	normal
Denial of service	No	normal
Logical design flaws	No	normal
Fake recharge vulnerability	No	normal
Short address attack	No	normal
Uninitialized storage pointer	No	normal
Frozen account bypass	No	normal
Uninitialized	No	normal
Reentry attack	No	normal

4.2 Risk audit details

4.2.1 Integer Overflow

- **Vulnerability description**

Integer overflow is generally divided into overflow and underflow. There are three types of integer overflow in smart contracts: multiplication overflow, addition overflow, and subtraction overflow. In the Solidity language, the integer type step size supported by the variable is incremented by 8, and it supports from uint8 to uint256, and int8 to int256. The integer specifies a fixed-size data type and is unsigned. For example, a uint8 type can only be stored Numbers in the range 0 to 2^8-1 , which is [0,255], a uint256 type can only store numbers in the range 0 to $2^{256}-1$. This means that an integer variable can only be represented by a certain range of numbers, and cannot exceed the specified range. Exceeding the value range expressed by the variable type will lead to an integer overflow vulnerability.

- **Audit result: passed**

4.2.2 Variable update

- **Vulnerability description**

Generally speaking, there will be multiple variable values in each contract code. Sometimes it may be because there are too many variables or written carelessly, some variables may not be updated in time. It may be a minor code problem, but for variables that involve funds In terms of calculation, in the eyes of an attacker, it is a big loophole that can steal funds.

- **Audit result: passed**

4.2.3 Administrator rights issue

- **Vulnerability description**

If the smart contract administrator has greater authority to operate the contract, if the administrator is manipulated by malicious personnel, it may cause abnormal capital loss and shake market stability.

- **Audit result: passed**

4.2.4 Numerical accuracy

- **Vulnerability description**

Solidity does not support floating-point type, nor does it fully support fixed-length floating-point type. The result of division operation will be rounded up. If there is a decimal, the part after the decimal point will be discarded, and only the integer part will be taken, for example, use 5 directly. Divide by 2, the result is 2. If the calculation result of the token is less than 1, for example, 4.9 tokens will be roughly equal to 4, which will cause a certain degree of loss of accuracy. Due to the economic properties of tokens, the loss of precision is equivalent to the loss of assets, so this will bring about the problem of accumulating in the frequently traded token.

- **Audit result: passed**

4.2.5 Default visibility

- **Vulnerability description**

In Solidity, the visibility of contract functions is public by default. Therefore, functions that do not specify any visibility can be called externally by the user. When a developer erroneously ignores the visibility specifier of a function that should be private, or a visibility specifier that can only be called within the contract itself, it will lead to serious vulnerabilities. In the first hack of the Parity multi-signature wallet, it was because the visibility of the function was not set, and the default was public, which led to the theft of a large amount of funds.

- **Audit result: passed**

4.2.6 tx.origin authentication

- **Vulnerability description**

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: passed**

4.2.7 Wrong constructor

- **Vulnerability description**

Before the 0.4.22 version of the solidity smart contract, all contracts and constructors had the same name. When writing a contract, if the constructor function name is not the same as the contract name, the contract will add a default constructor function, and the constructor function set by yourself will be treated as a normal function, causing your original contract settings to not execute as expected, which may cause terrible Consequences, especially if the constructor is performing a privileged operation.

- **Audit result: passed**

4.2.8 Unverified return value

- **Vulnerability description**

There are three methods to send tokens to an address in Solidity: `transfer()`, `send()`, `call.value()`. The difference between them is that when the `transfer` function fails to send, it throws an exception `throw`, rolls back the transaction status, and costs 2300 gas; when the `send` function fails to send, it returns `false` and costs 2300 gas; when the `call.value` method fails to send, it returns `false`, and the call costs all gas. Will lead to the risk of re-entry attacks. If the `send` or `call.value` method is used in the contract code to send the token without checking the method return value, if an error occurs, the contract will continue to execute the subsequent code, which will lead to the expected result.

- **Audit result: passed**

4.2.9 Insecure random number

- **Vulnerability description**

All transactions on the blockchain are deterministic state transition operations without uncertainty, which ultimately means that there is no source of entropy or randomness in the blockchain ecosystem. So there is no random number function like `rand()` in Solidity. Many developers use future block variables, such as block hash value, timestamp, block height, or gas upper limit to generate random numbers. These quantities are controlled by the miners, so they are not truly random. , So using past or current block variables to generate random numbers may lead to destructive vulnerabilities.

- **Audit result: passed**

4.2.10 Timestamp dependent

- **Vulnerability description**

In the blockchain, the data block timestamp (`block.timestamp`) is used in various applications, such as the function of random numbers, the locking of funds for a period of time, and the conditional statements of various state changes related to time. Miners have the ability to adjust the timestamp according to their needs. For example, `block.timestamp` or the alias `now` can be manipulated by the miners. If the wrong block timestamp is used in the smart contract, this can lead to serious vulnerabilities. If the contract is not particularly concerned about the miner's manipulation of the block timestamp, this may be unnecessary, but this should be paid attention to when developing the contract.

- **Audit result: passed**

4.2.11 Transaction order dependence

- **Vulnerability description**

In the blockchain, the miners choose the transaction with the highest transaction fee and pack it into the block. Since the transaction information in the block is public, the attacker can observe whether there are transactions in the transaction pool that may contain a solution to the problem, modify or revoke the attacker's authority or change the state of the contract that is unfavorable to the attacker. Then, the attacker can obtain data from this transaction and create a higher-level transaction `gasPrice` and include its transaction in a block before the original, which will preempt the original transaction solution.

- **Audit result: passed**

4.2.12 Delegatecall

- **Vulnerability description**

In Solidity, the `delegatecall` function is a standard message calling method, but the code in the target address will run in the environment of the calling contract, that is, keep `msg.sender` and `msg.value` unchanged. This function supports the implementation of the library, and developers can create reusable code for future contracts. The code in the library itself can be safe and flawless, but when running in another application environment, new vulnerabilities may occur, so using the `delegatecall` function may cause unexpected code execution.

- **Audit result: passed**

4.2.13 Call

- **Vulnerability description**

The call function is similar to the delegatecall function. They are both low-level functions provided by the smart contract writing language Solidity to interact with external contracts or libraries. However, when the call function method is used to process the call to the external standard information of the contract, the code is in the external contract/ Run in a functional environment. When using this type of function, it is necessary to determine the security of the calling parameters. It is recommended to use it with caution. Attackers can easily borrow the identity of the current contract to perform other malicious operations, leading to serious vulnerabilities.

- **Audit result: passed**

4.2.14 Denial of service

- **Vulnerability description**

There are a wide range of reasons for denial-of-service attacks, and its purpose is to allow users to make the contract unable to function normally for a period of time or permanently under certain circumstances, including malicious behavior when acting as a transaction receiver, and artificially increasing the gas required for computing functions. Leading to gas exhaustion, abusing access control to access the private components of the contract, the owner with privileges in the contract is modified, based on external calls, using obfuscation, etc. can lead to denial of service attacks.

- **Audit result: passed**

4.2.15 Logical design flaws

- **Vulnerability description**

In smart contracts, the special functions designed by developers for their own contracts are intended to stabilize the market value of tokens or the life of the project, and increase the highlights of the project. However, the more complex the system, the more likely it is to make mistakes. It is precisely in these logic and In the function, a slight error may lead to a serious deviation between the whole logic and the expectation, leaving fatal hidden dangers, such as logic judgment errors, function implementation and design inconsistency etc.

- **Audit result: passed**

4.2.16 Fake recharge vulnerability

- **Vulnerability description**

The success or failure (true or false) of token transaction receipt depends on whether an exception is thrown during transaction execution (for example, the mechanism of require / assert / reverse / throw is used). When the user calls the transfer function of the token contract for transfer, if the transfer function runs normally and does not throw an exception, whether the transfer transaction is successful or not, the receipt status of the transaction is success, that is true. Then, the transfer function of some token contracts checks the balance of the transfer initiator (MSG. Sender) by using if judgment. When balances [MSG. Sender] < 0_When value, it enters the else logic part and returns false, and finally no exception is thrown, but the transaction receipt is successful. Then we think that only if / else, a mild judgment method, is an imprecise coding method in the scenario of sensitive functions such as transfer, which will lead to false recharge loopholes in related centralized exchanges, centralized wallets and token contracts.

- **Audit result: passed**

4.2.17 Short address attack

- **Vulnerability description**

In the solid smart contract, when parameters are passed to the smart contract, the parameters will be coded according to ABI specification. EVM running attackers send encoding parameters shorter than expected. For example, when transferring money in an exchange or a wallet, you need to send the transfer address and the transfer amount value. An attacker can send a 19 byte address instead of a standard 20 byte address. In this case, EVM will fill 0 in the end of the encoding parameter to make up the expected length, which will lead to the overflow of the final transfer amount parameter value and change the original transfer amount.

- **Audit result: passed**

4.2.18 Uninitialized storage pointer

- **Vulnerability description**

EVM uses both storage and memory to store variables. Local variables in functions are stored in storage or memory by default according to their types. In the working mode of solid, state variables are stored in slots of contracts according to the order in which they appear in contracts, Uninitialized local storage variables may point to other unexpected storage variables in the contract, resulting in intentional or unintentional vulnerabilities.

- **Audit result: passed**

4.2.19 Frozen account bypass

- **Vulnerability description**

In the transfer operation code of the contract, it detects whether the logic function of checking the freezing status of the transfer account exists in the contract code, and if the transfer account has been frozen, whether it can be bypassed.

- **Audit result: passed**

4.2.20 Uninitialized

- **Vulnerability description**

The initialize function in the contract can be called by other attackers before grabbing owner, so as to initialize governor.

- **Audit result: passed**

4.2.21 Reentry attack

- **Vulnerability description**

The attacker constructs a contract containing malicious code at the external address of fallback function. When the contract sends tokens to this address, it will call malicious code. When the call. Value() function in solid is used to send tokens, it will consume all the gas it receives. Therefore, when the call. Value() function is called to send tokens before the actual reduction of the sender's account balance, a reentry attack will occur. Due to the reentry vulnerability, the Dao attack is well-known.

- **Audit result: passed**

5. Security Audit Tool

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

Disclaimer:

Noneage Technology only issues a report and assumes corresponding responsibilities for the facts that occurred or existed before the issuance of this report, Since the facts that occurred after the issuance of the report cannot determine the security status of the smart contract, it is not responsible for this. Noneage Technology conducts security audits on the security audit items in the project agreement, and is not responsible for the project background and other circumstances, The subsequent on-chain deployment and operation methods of the project party are beyond the scope of this audit. This report only conducts a security audit based on the information provided by the information provider to Noneage at the time the report is issued, If the information of this project is concealed or the situation reflected is inconsistent with the actual situation, Noneage Technology shall not be liable for any losses and adverse effects caused thereby.

There are risks in the market, and investment needs to be cautious. This report only conducts security audits and results announcements on smart contract codes, and does not make investment recommendations and basis.



Email : support@noneage.com

Site : www.noneage.com

Weibo : weibo.com/noneage

