Smart Contract Audit Report

C10Token

Audit Result: Passed

Version Description

Reviser	Revisions	Revision Date	Version	Reviewer
Wang Hailong	Create document	14-22-2019	V1.0	Sui Xin

Document Information

Name	Version	Document ID	Classification Level
Smart Contract Audit Report	V1.0	0xee2de48b(C10Token.sol) 0x412798ed(InvictusWhitelist.sol)	Private

Copyright Notice

Table of Contents

Ⅰ、Review

1.1 Audit Background

Cheetah Mobile Security team conducted smart contract audit for C10Token on April 22, 2019 . This report presents the audit details and results.

(**Disclaimer**: Cheetah Mobile Security only issues the report based on the vulnerabilities existed before the issuance of the report, and bears corresponding responsibility. As for the facts that occur or exist after the issuance of the report, whose impacts on the project cannot be determined, Cheetah Mobile Security is not responsible for the consequence. The security audit analysis and other contents of this report are based only on the documents and information provided by the information provider to Cheetah Mobile Security as of the date of issuance of the report. The information provider is obliged to ensure that there is not missing, falsified, deletion or concealment of the information provided. If present, Cheetah Mobile Security shall not be liable for any loss or adverse effect caused by this situation.)

1.2 Audit Result

Audit Result	Auditor	Reviewer
Passed	Wang Hailong	Sui Xin

1.3 Basic Information

- Contract name：

  C10Token

- Contract type：

  Token contract, token name: C10Token Hedged(C10)

II、 Contract Code Vulnerability Analysis

2.1 Contract Code Vulnerability Levels

The number of contract vulnerabilities is counted by level as follows：

High Risk	Medium Risk	Low Risk
0	0	0

2.2 Contract Code Vulnerability Distribution

Not found

2.3 Contract Code Audit Items and Results

(Other unknown vulnerabilities are not included in the scope of responsibility of this audit)

Audit Method	Audit Class	Audit Subclass	Audit Result
Offensive/ Defensive Audit	Arithmetic Safety	Integer Overflow	Passed
		Integer Underflow	Passed
		Operation Precision	Passed
	Competitive Competition	Reentrancy	Passed
		Transaction Ordering Dependence	Passed
	Access Control	Privilege Vulnerability	Passed
		Overprivileged Audit	Passed （There is a logic in the contract that the project party represents the investor to destroy the token (see the burnForParticipant function for details), and investors are invited to understand the function. It is recommended that the project party give a description of the function.）

	Security Design	Security Module Usage	Passed
		Compiler Version Security	Passed
		Hard Coded Address Security	Passed
		Sensitive Functions (fallback/call/tx.origin) Security	Passed
		Function Return Value	Passed
	Denial of Service	-	Passed
	Gas Optimazation	-	Passed
	Design Logic	-	Passed

III、 Audit Details

3.1 Arithmetic Safety Audit

The arithmetic security audit is divided into three parts: integer overflow audit, integer underflow audit and operation precision audit.

3.1.1 Integer Overflow Audit【Passed】

Solidity can handle 256 bits of data at most. When the maximum number increases, it will overflow. If the integer overflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated, resulting in serious capital risk.

Audit Result: The code meets the specification.

Security Recommendation: No

3.1.2 Integer Underflow Audit【Passed】

Solidity can handle 256 bits of data at most. When the minimum number decreases, it will overflow. If the integer underflow occurs in the transfer logic, it will make the amount of transfer funds miscalculated and lead to serious capital risk.

Audit Result: The code meets the specification.

Security Recommendation: No

3.1.3 Operation Precision Audit【Passed】

Solidity performs type coercion in the process of multiplication and division. If the precision risk is included in the operation of capital variable, it will lead to user transfer logic error and capital loss.

Audit Result: The code meets the specification.

Security Recommendation: No

3.2 Competitive Competition Audit

The competitive competition audit is divided into two parts: reentrancy audit and transaction ordering dependence audit. With competitive vulnerabilities, an attacker can modify the output of a program by adjusting the execution process of transactions with a certain probability.

3.2.1 Reentrancy Audit【Passed】

Reentrancy occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete. For a function, this means that the contract state may change in the middle of its execution as a result of a call to an untrusted contract or the use of a low level function with an external address.

Audit Result： The code meets the specification.

Security Recommendation： No

3.2.2 Transaction Ordering Dependence Audit【Passed】

Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the Ethereum blockchain is public, everyone can see the contents of others' pending transactions. This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution. If developers of smart contracts are not careful, this situation can lead to practical and devastating front-running attacks.

Audit Result： The code meets the specification.

Security Recommendation: No

3.3 Access Control Audit

Access control audit is divided into two parts: privilege vulnerability audit and overprivileged audit.

3.3.1 Privilege Vulnerability Audit【Passed】

Smart contracts with privilege vulnerability, attackers can weigh their own accounts to gain higher execution privileges.

Audit Result: The code meets the specification.

Security Recommendation: No

3.3.2 Overprivileged Audit【Passed】

Overprivileged auditing focuses on whether there are special user privileges in audit contracts, such as allowing a user to unlimitly mine tokens.

Audit Result: The code meets the specification.

Security Recommendation: No

3.4 Security Design Audit

Security design audit is divided into five parts: security module usage, compiler version security, hard-coded address security, sensitive function usage security and function return value security.

3.4.1 Security Module Usage【Passed】

The security module usage audit whether the smart contract uses the SafeMath library function provided by OpenZepplin to avoid overflow vulnerabilities; if it does not, whether the transfer amount is strictly checked during the execution.

Audit Result: The code meets the specification.

Security Recommendation: No

3.4.2 Compiler Version Security Audit【Passed】

Compiler version security focuses on whether the smart contract explicitly indicates the compiler version and whether the compiler version used is too low to throw an exception.

Audit Result: The code meets the specification.

Security Recommendation: No

3.4.3 Hard Coded Address Security Audit【Passed】

Hard-coded address security audit static addressed in the smart contract to check whether there is an exception to the external contract, thus affecting the execution of this contract.

Audit Result： The code meets the specification.

Security Recommendation： No

3.4.4 Sensitive Functions Audit【Passed】

Sensitive functions audit checks whether the smart contract uses the unrecommandation functions such as fallback, call and tx.origin.

Audit Result： The code meets the specification.

Security Recommendation： No

3.4.5 Function Return Value Audit【Passed】

Function return value audit mainly analyzes whether the function correctly throws an exception, correctly returns to the state of the transaction.

Audit Result： The code meets the specification.

Security Recommendation： No

3.5 Denial of Service Audit【Passed】

Denial of service attack sometimes can make the smart contract offline forever by maliciously behaving when being the recipient of a transaction, artificially increasing the gas necessary to compute a function, abusing access controls to access private components of smart contracts, taking advantage of mixups and negligence and so on.

Audit Result： The code meets the specification.

Security Recommendation： No

3.6 Gas Optimization Audit【Passed】

If the computation of a function in a smart contract is too complex, such as the batch transfer to a variable-length array through a loop, it is very easy to cause the gas fee beyond the block's gas Limit resulting in transaction execution failure.

Audit Result： The code meets the specification.

Security Recommendation： No

3.7 Design Logic Audit【Passed】

In addition to vulnerabilities, there are logic problems in the process of code implementation, resulting in abnormal execution results.

Audit Result： The code meets the specification.

Security Recommendation： No

Appendix I: Contract Code Audit Details

1、C10Token.sol

```
pragma solidity ^0.5.6;//Cheetah Mobile Security//Specifying the
compiled version conforming to the security coding specification.


import
"./openzeppelin-solidity/contracts/token/ERC20/ERC20Detailed.sol";
import "./openzeppelin-solidity/contracts/token/ERC20/ERC20.sol";
import
"./openzeppelin-solidity/contracts/token/ERC20/ERC20Burnable.sol";
import "./openzeppelin-solidity/contracts/token/ERC20/SafeERC20.sol";
import
"./openzeppelin-solidity/contracts/access/roles/MinterRole.sol";
import "./openzeppelin-solidity/contracts/lifecycle/Pausable.sol";
import "./openzeppelin-solidity/contracts/ownership/Ownable.sol";
import "./openzeppelin-solidity/contracts/math/SafeMath.sol";
import "./InvictusWhitelist.sol";



/**
 * Contract for CRYPTO10 Hedged (C10) fund.
 *
 */
contract C10Token is ERC20, ERC20Detailed, ERC20Burnable, Ownable,
Pausable, MinterRole {
```

```
    using SafeERC20 for ERC20;

    using SafeMath for uint256;



    // Maps participant addresses to the eth balance pending token issuance

    mapping(address => uint256) public pendingBuys;

    // The participant accounts waiting for token issuance

    address[] public participantAddresses;



    // Maps participant addresses to the withdrawal request

    mapping (address => uint256) public pendingWithdrawals;

    address payable[] public withdrawals;



    uint256 private minimumWei = 50 finney;//Cheetah Mobile
Security//Specify the minimum amount of investment

    uint256 private fees = 5;  // 0.5% , or 5/1000

    uint256 private minTokenRedemption = 1 ether;

    uint256 private maxAllocationsPerTx = 50;

    uint256 private maxWithdrawalsPerTx = 50;

    Price public price;



    address public whitelistContract; //Cheetah Mobile Security//Define
a whitelist contract address

        //Cheetah Mobile Security//Define the Price data type

    struct Price {

        uint256 numerator;

        uint256 denominator;
```

```
    }
    //Cheetah Mobile Security//Define 8 events
       event PriceUpdate(uint256 numerator, uint256 denominator);
    event AddLiquidity(uint256 value);
    event RemoveLiquidity(uint256 value);
    event DepositReceived(address indexed participant, uint256 value);
    event TokensIssued(address indexed participant, uint256 amountTokens,
uint256 etherAmount);
    event WithdrawRequest(address indexed participant, uint256
amountTokens);
    event Withdraw(address indexed participant, uint256 amountTokens,
uint256 etherAmount);
    event TokensClaimed(address indexed token, uint256 balance);
    //Cheetah Mobile Security//The constructor is used correctly
constructor (uint256 priceNumeratorInput, address
whitelistContractInput)
       ERC20Detailed("Crypto10 Hedged", "C10", 18)
       ERC20Burnable()
       Pausable() public {
          price = Price(priceNumeratorInput, 1000);
          require(priceNumeratorInput > 0, "Invalid price numerator");
          //Cheetah Mobile Security//Legitimacy judgment on the incoming
whitelistContractInput
          require(whitelistContractInput != address(0), "Invalid
whitelist address");
          whitelistContract = whitelistContractInput;    }
```

```
    /**
     * @dev fallback function that buys tokens if the sender is whitelisted.
     */
    function () external payable {
        buyTokens(msg.sender);
    }

    /**
     * @dev Explicitly buy via contract.
     */
    function buy() external payable {
        buyTokens(msg.sender);
    }

    /**
     * Sets the maximum number of allocations in a single transaction.
     * @dev Allows us to configure batch sizes and avoid running out of
gas.
     */
    function setMaxAllocationsPerTx(uint256 newMaxAllocationsPerTx)
external onlyOwner {
        require(newMaxAllocationsPerTx > 0, "Must be greater than 0");
        maxAllocationsPerTx = newMaxAllocationsPerTx;
    }
```

```
    /**
     * Sets the maximum number of withdrawals in a single transaction.
     * @dev Allows us to configure batch sizes and avoid running out of
gas.
     */
    function setMaxWithdrawalsPerTx(uint256 newMaxWithdrawalsPerTx)
external onlyOwner {
        require(newMaxWithdrawalsPerTx > 0, "Must be greater than 0");
        maxWithdrawalsPerTx = newMaxWithdrawalsPerTx;
    }

    /// Sets the minimum wei when buying tokens.
    function setMinimumBuyValue(uint256 newMinimumWei) external
onlyOwner {
        require(newMinimumWei > 0, "Minimum must be greater than 0");
        minimumWei = newMinimumWei;
    }

    /// Sets the minimum number of tokens to redeem.
    function setMinimumTokenRedemption(uint256 newMinTokenRedemption)
external onlyOwner {
        require(newMinTokenRedemption > 0, "Minimum must be greater than
0");
        minTokenRedemption = newMinTokenRedemption;
    }
```

```
    /// Updates the price numerator.
    function updatePrice(uint256 newNumerator) external onlyMinter {
        require(newNumerator > 0, "Must be positive value");

        price.numerator = newNumerator;

        allocateTokens();
        processWithdrawals();
        emit PriceUpdate(price.numerator, price.denominator);
    }

    /// Updates the price denominator.
    function updatePriceDenominator(uint256 newDenominator) external
onlyMinter {
        require(newDenominator > 0, "Must be positive value");

        price.denominator = newDenominator;
    }

    /**
     * Whitelisted token holders can request token redemption, and withdraw
ETH.
     * @param amountTokensToWithdraw The number of tokens to withdraw.
     * @dev withdrawn tokens are burnt.
     */
```

```
    //Cheetah Mobile Security//Function modifier is used correctly
function requestWithdrawal(uint256 amountTokensToWithdraw) external
whenNotPaused onlyWhitelisted {

        address payable participant = msg.sender;
        require(balanceOf(participant) >=
amountTokensToWithdraw,"Cannot withdraw more than balance held");
        require(amountTokensToWithdraw >= minTokenRedemption, "Too few
tokens");
        //Cheetah Mobile Security//Destroy tokens
        burn(amountTokensToWithdraw);

        uint256 pendingAmount = pendingWithdrawals[participant];
        if (pendingAmount == 0) {
            withdrawals.push(participant); //Cheetah Mobile
Security//Add participant to withdrawals
        }
        pendingWithdrawals[participant] =
pendingAmount.add(amountTokensToWithdraw);//Cheetah Mobile
Security//update pendingWithdrawals[participant]
        emit WithdrawRequest(participant, amountTokensToWithdraw);
    }

    /// Allows owner to claim any ERC20 tokens.
    //Cheetah Mobile Security//Function modifier is used correctly
    function claimTokens(ERC20 token) external payable onlyOwner {
```

```
        //Cheetah Mobile Security//Legitimate judgment on incoming
parameters
        require(address(token) != address(0), "Invalid address");
        uint256 balance = token.balanceOf(address(this));
        //Cheetah Mobile Security//Call the token's transfer function
        token.transfer(owner(), token.balanceOf(address(this)));
        emit TokensClaimed(address(token), balance);
    }

    /**
     * @dev Allows the owner to burn a specific amount of tokens on a
participant's behalf.
     * @param value The amount of tokens to be burned.
     */
    //Cheetah Mobile Security//The project party can destroy the token
on behalf of the investor
        function burnForParticipant(address account, uint256 value)
public onlyOwner {
        _burn(account, value);
    }

    /**
     * @dev Function to mint tokens when not paused.
     * @param to The address that will receive the minted tokens.
     * @param value The amount of tokens to mint.
     * @return A boolean that indicates if the operation was successful.
```

```
     */
    //Cheetah Mobile Security//Function modifier is used correctly
        function mint(address to, uint256 value) public onlyMinter
whenNotPaused returns (bool) {
        _mint(to, value);

        return true;
    }


    /// Adds liquidity to the contract, allowing anyone to deposit ETH
    function addLiquidity() public payable {
        require(msg.value > 0, "Must be positive value");
        emit AddLiquidity(msg.value);
    }


    /// Removes liquidity, allowing managing wallets to transfer eth to
the fund wallet.
    function removeLiquidity(uint256 amount) public onlyOwner {
        require(amount <= address(this).balance, "Insufficient
balance");

        msg.sender.transfer(amount);
        emit RemoveLiquidity(amount);
    }
```

```
    /// Allow the owner to remove a minter     function
removeMinter(address account) public onlyOwner {
        require(account != msg.sender, "Use renounceMinter");
        _removeMinter(account);
    }

    /// Allow the owner to remove a pauser
    function removePauser(address account) public onlyOwner {
        require(account != msg.sender, "Use renouncePauser");
        _removePauser(account);
    }

    /// returns the number of withdrawals pending.
    //Cheetah Mobile Security//The view modifier is used correctly
    function numberWithdrawalsPending() public view returns (uint256) {
        return withdrawals.length;
    }

    /// returns the number of pending buys, waiting for token issuance.
    //Cheetah Mobile Security//The view modifier is used correctly
    function numberBuysPending() public view returns (uint256) {
        return participantAddresses.length;
    }

    /**
```

```
     * First phase of the 2-part buy, the participant deposits eth and
waits
     * for a price to be set so the tokens can be minted.
     * @param participant whitelisted buyer.
     */
    function buyTokens(address participant) internal whenNotPaused
onlyWhitelisted {
        //Cheetah Mobile Security//Determine the legality of incoming
parameters
        assert(participant != address(0));

        // Ensure minimum investment is met
        //Cheetah Mobile Security//Determine if msg.value meets the
minimum investment criteria
        require(msg.value >= minimumWei, "Minimum wei not met");

        uint256 pendingAmount = pendingBuys[participant];
        if (pendingAmount == 0) {
            participantAddresses.push(participant); //Cheetah Mobile
Security//Add new elements to the participantAddresses
        }

        // Increase the pending balance and wait for the price update
        //Cheetah Mobile Security//Use SafeMath to prevent integer
overflows, in line with security practices
        pendingBuys[participant] = pendingAmount.add(msg.value);
```

```
        emit DepositReceived(participant, msg.value);
    }

    /// Internal function to allocate token.
    function allocateTokens() internal {
        uint256 numberOfAllocations = participantAddresses.length <=
maxAllocationsPerTx ?
            participantAddresses.length : maxAllocationsPerTx;

        address payable ownerAddress = address(uint160(owner()));
        for (uint256 i = numberOfAllocations; i > 0; i--) {
            address participant = participantAddresses[i - 1];
            uint256 deposit = pendingBuys[participant];
            //Cheetah Mobile Security//Use SafaMath for integer arithmetic,
in line with security practices
            uint256 feeAmount = deposit.mul(fees) / 1000;
            uint256 balance = deposit.sub(feeAmount);

            uint256 newTokens = balance.mul(price.numerator) /
price.denominator;
            pendingBuys[participant] = 0;
            participantAddresses.pop();

            ownerAddress.transfer(feeAmount);
            //Cheetah Mobile Security//Call coinage operation
```

```
            mint(participant, newTokens);

            emit TokensIssued(participant, newTokens, balance);

        }

    }


    /// Internal function to process withdrawals.

    function processWithdrawals() internal {

        uint256 numberOfWithdrawals = withdrawals.length <=
maxWithdrawalsPerTx ? withdrawals.length : maxWithdrawalsPerTx;


        address payable ownerAddress = address(uint160(owner()));

        for (uint256 i = numberOfWithdrawals; i > 0; i--) {

            address payable participant = withdrawals[i - 1];

            uint256 tokens = pendingWithdrawals[participant];


            assert(tokens > 0); // participant must have requested a
withdrawal

            //Cheetah Mobile Security//Integer operation using SafeMath,
in line with smart contract security development practices

            uint256 withdrawValue = tokens.mul(price.denominator) /
price.numerator;

            //Cheetah Mobile Security//Set to 0

            pendingWithdrawals[participant] = 0;

            //Cheetah Mobile Security//Remove

            withdrawals.pop();
```

```
            if (address(this).balance >= withdrawValue) {
                //Cheetah Mobile Security//Integer operation using
SafeMath complies with safety contract development practices
                uint256 feeAmount = withdrawValue.mul(fees) / 1000;
                uint256 balance = withdrawValue.sub(feeAmount);
                //Cheetah Mobile Security//Give participant turn balance
                participant.transfer(balance);
                //Cheetah Mobile Security//Turn ownerAmount to feelAmount
                ownerAddress.transfer(feeAmount);

                emit Withdraw(participant, tokens, balance);
            }
            else {
                mint(participant, tokens);
                emit Withdraw(participant, tokens, 0); // indicate a failed
withdrawal
            }
        }
    }

    modifier onlyWhitelisted() {

require(InvictusWhitelist(whitelistContract).isWhitelisted(msg.sender)
, "Must be whitelisted");
        _;
    }
```

```
}

2、InvictusWhitelist.sol

pragma solidity ^0.5.6; //Cheetah Mobile Security//Specifying the

compiled version conforming to the security coding specification.


import "./openzeppelin-solidity/contracts/ownership/Ownable.sol";

import

"./openzeppelin-solidity/contracts/access/roles/WhitelistedRole.sol";


/**

 * Manages whitelisted addresses.

 *

 */

contract InvictusWhitelist is Ownable, WhitelistedRole {

    //Cheetah Mobile Security//The constructor is used correctly

        constructor ()

        WhitelistedRole() public {

    }


    /// @dev override to support legacy name

    function verifyParticipant(address participant) public

onlyWhitelistAdmin {

        if (!isWhitelisted(participant)) {

            addWhitelisted(participant);

        }

    }
```

```
    /// Allow the owner to remove a whitelistAdmin
    function removeWhitelistAdmin(address account) public onlyOwner {
        //Cheetah Mobile Security//Judge the incoming parameters
        require(account != msg.sender, "Use renounceWhitelistAdmin");
        _removeWhitelistAdmin(account);
    }
}
```

3、openzepplin-solidity table of Contents

The code in this directory is taken from a third-party smart contract security library.https://github.com/OpenZeppelin/openzeppelin-solidity

We have audited the code used in this contract and found no security issues. Since this part of the code is relatively large and not listed, it is hereby stated.

Cheetah Mobile Security

audit@cmcm.com