知道创宇

密级 ★ | 项目组公开

# 智能合约审计报告

安全状态

## 安全

★ ★ ★ ★ ★

主测人：知道创宇云安全服务团队

## 文档信息

| 文档名称 | 文档 MD5 | 文档版本 |
| --- | --- | --- |
| JCT 合约审计报告 | 26a49f41f9df6447f5bb6239e9223392 | V1.0 |

## 版本说明

| 修订人 | 修订内容 | 修订时间 | 保密级别 |
| --- | --- | --- | --- |
| 知道创宇云安全服务团队 | JCT 合约审计报告 | 2020.09.23 | 项目组公开 |

## 版权说明

# 目录

# 1. 综述

本次报告有效测试时间是从 2020 年 9 月 20 日开始到 2020 年 9 月 23 日结束，在此期间针对 JCT 智能合约代码的安全性和规范性进行审计并以此作为报告统计依据。

此次测试中，知道创宇工程师对智能合约的常见漏洞（见第三章节）进行了全面的分析，未发现安全问题，综合评定为**通过**。

**本次智能合约安全审计结果：通过**

由于本次测试过程在非生产环境下进行，所有代码均为最新备份，测试过程均与相关接口人进行沟通，并在操作风险可控的情况下进行相关测试操作，以规避测试过程中的生产运营风险、代码安全风险。

**本次测试的目标信息**

| | |
|---|---|
| **Token 名称** | **Jadeite & co Token (JCT)** |
| **代币地址** | http://explorer.moac.io/token/<br>0x1ef191730c0094c8bba0c18818c2938da9909527 |
| **代码类型** | **代币代码** |
| **代码语言** | **Solidity** |
| **合约地址** | **合约源码** |

**合约文件及哈希**

| **合约文件** | **MD5 值** |
|---|---|

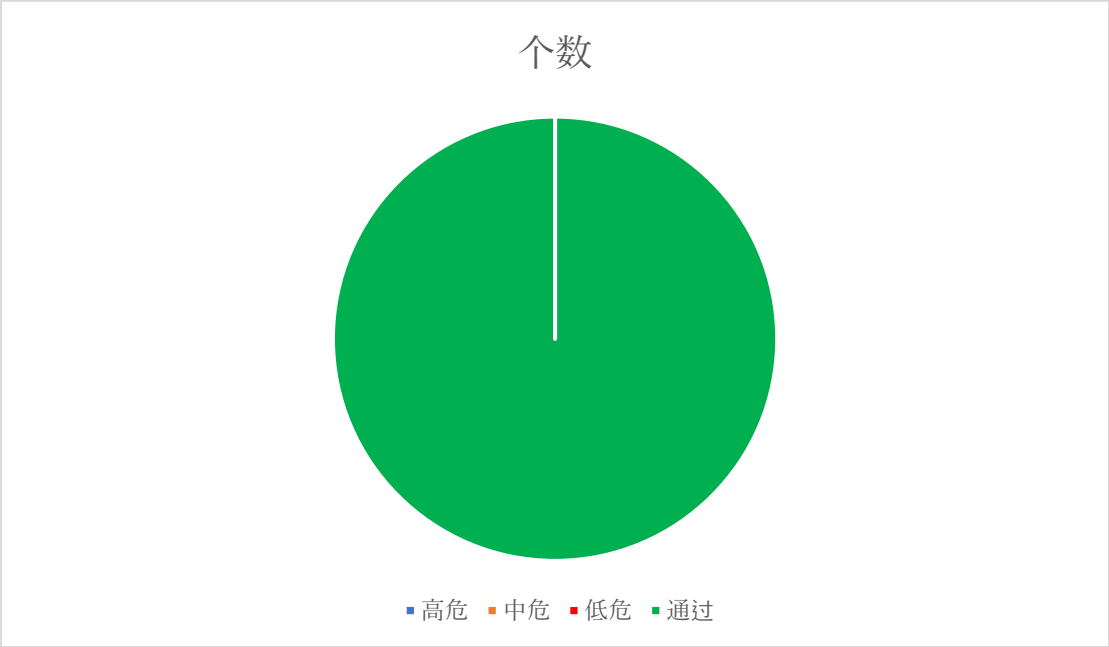
| | |
|---|---|
| **true_erc20_JCT_new_2020908_good.sol** | **79f52cca6f2edcec592217523a3ec744** |

# 2. 代码漏洞分析

## 2.1. 漏洞等级分布

本次漏洞风险按等级统计：

| 漏洞风险等级个数统计表 | | | |
|---|---|---|---|
| 高危 | 中危 | 低危 | 通过 |
| 0 | 0 | 0 | 11 |




## 2.2. 审计结果汇总说明

| 审计结果 | | | |
|---|---|---|---|
| 测试序号 | 测试内容 | 状态 | 描述 |
| 1 | 重入攻击检测 | 通过 | 经检测，不存在该安全问题。 |
| 2 | 数值溢出检测 | 通过 | 经检测，不存在该安全问题。 |
| 3 | 访问控制缺陷检测 | 通过 | 经检测，不存在该安全问题。 |

| | | | |
|---|---|---|---|
| 4 | 未验证返回值的调用 | 通过 | 经检测，不存在该安全问题。 |
| 5 | 错误使用随机数检测 | 通过 | 经检测，不存在该安全问题。 |
| 6 | 事务顺序依赖检测 | 通过 | 经检测，不存在该安全问题。 |
| 7 | 拒绝服务攻击检测 | 通过 | 经检测，不存在该安全问题。 |
| 8 | 逻辑设计缺陷检测 | 通过 | 经检测，不存在该安全问题。 |
| 9 | 假充值漏洞检测 | 通过 | 经检测，不存在该安全问题。 |
| 10 | 增发代币漏洞检测 | 通过 | 经检测，不存在该安全问题。 |
| 11 | 冻结账户绕过检测 | 通过 | 经检测，不存在该安全问题。 |

# 3. 代码审计结果分析

## 3.1. 重入攻击检测【通过】

重入漏洞是最著名的以太坊智能合约漏洞，曾导致了以太坊的分叉（The DAO hack）。

Solidity 中的 call.value()函数在被用来发送 Ether 的时候会消耗它接收到的所有 gas ,当调用 call.value()函数发送 Ether 的操作发生在实际减少发送者账户的余额之前时，就会存在重入攻击的风险。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 3.2. 数值溢出检测【通过】

智能合约中的算数问题是指整数溢出和整数下溢。

Solidity 最多能处理 256 位的数字（2^256-1），最大数字增加 1 会溢出得到 0。同样，当数字为无符号类型时，0 减去 1 会下溢得到最大数字值。

整数溢出和下溢不是一种新类型的漏洞，但它们在智能合约中尤其危险。溢出情况会导致不正确的结果，特别是如果可能性未被预期，可能会影响程序的可靠性和安全性。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 3.3. 访问控制检测【通过】

访问控制缺陷是所有程序中都可能存在的安全风险，智能合约也同样会存在类似问题，著名的 Parity Wallet 智能合约就受到过该问题的影响。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 3.4. 返回值调用验证【通过】

此问题多出现在和转币相关的智能合约中，故又称作静默失败发送或未经检查发送。

在 Solidity 中存在 transfer()、send()、call.value()等转币方法，都可以用于向某一地址发送 Ether，其区别在于： transfer 发送失败时会 throw，并且进行状态回滚；只会传递 2300gas 供调用，防止重入攻击；send 发送失败时会返回 false；只会传递 2300gas 供调用，防止重入攻击；call.value 发送失败时会返回 false；传递所有可用 gas 进行调用（可通过传入 gas_value 参数进行限制），不能有效防止重入攻击。

如果在代码中没有检查以上 send 和 call.value 转币函数的返回值，合约会继续执行后面的代码，可能由于 Ether 发送失败而导致意外的结果。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 3.5. 错误使用随机数【通过】

智能合约中可能需要使用随机数，虽然 Solidity 提供的函数和变量可以访问明显难以预测的值，如 block.number 和 block.timestamp，但是它们通常或者比看

起来更公开，或者受到矿工的影响，即这些随机数在一定程度上是可预测的，所以恶意用户通常可以复制它并依靠其不可预知性来攻击该功能。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 3.6. 事务顺序依赖【通过】

由于矿工总是通过代表外部拥有地址（EOA）的代码获取 gas 费用，因此用户可以指定更高的费用以便更快地开展交易。由于以太坊区块链是公开的，每个人都可以看到其他人未决交易的内容。这意味着，如果某个用户提交了一个有价值的解决方案，恶意用户可以窃取该解决方案并以较高的费用复制其交易，以抢占原始解决方案。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无

## 3.7. 拒绝服务攻击【通过】

在以太坊的世界中，拒绝服务是致命的，遭受该类型攻击的智能合约可能永远无法恢复正常工作状态。导致智能合约拒绝服务的原因可能有很多种，包括在作为交易接收方时的恶意行为，人为增加计算功能所需 gas 导致 gas 耗尽，滥用访问控制访问智能合约的 private 组件，利用混淆和疏忽等等。

**检测结果：**经检测，智能合约代码中不存在该安全问题。

**安全建议：**无。

## 3.8. 逻辑设计缺陷【通过】

检测智能合约代码中与业务设计相关的安全问题。

**检测结果**：经检测，智能合约代码中不存在该安全问题。

**安全建议**：无。

## 3.9. 假充值漏洞【通过】

在代币合约的 transfer 函数对转账发起人(msg.sender)的余额检查用的是 if 判断方式，当 balances[msg.sender] < value 时进入 else 逻辑部分并 return false，最终没有抛出异常，我们认为仅 if/else 这种温和的判断方式在 transfer 这类敏感函数场景中是一种不严谨的编码方式。

**检测结果**：经检测，智能合约代码中不存在该安全问题。

**安全建议**：无。

## 3.10. 增发代币漏洞【通过】

检测在初始化代币总量后，代币合约中是否存在可能使代币总量增加的函数。

**检测结果**：经检测，智能合约代码中不存在该安全问题。

**安全建议**：无。

## 3.11. 冻结账户绕过【通过】

检测代币合约中在转移代币时，是否存在未校验代币来源账户、发起账户、目标账户是否被冻结的操作。

**检测结果**：经检测，智能合约代码中不存在该安全问题。

**安全建议**：无。

# 4. 附录 A：合约代码

**本次测试代码来源：以下源码由项目方提供**

```
pragma solidity ^0.4.18; //knownsec //指定编译器版本，符合推荐做法
/**
 * Math operations with safety checks
 */
contract SafeMath { //knownsec//使用了安全的数值操作函数，不存在数值溢出风险
    function safeMul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }
        uint256 c = a * b;
        require(c / a == b);
        return c;
    }
    function safeDiv(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b > 0);
        uint256 c = a / b;
        require(a == b * c + a % b);
        return c;
    }
    function safeSub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a);
        return a - b;
    }
    function safeAdd(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c>=a && c>=b);
        return c;
    }
}
contract Token is SafeMath{
    function balanceOf(address _owner) public constant returns (uint256 balance);
    function transfer(address _to, uint256 _value) public returns (bool success);
```

```
        function transferFrom(address _from, address _to, uint256 _value) public returns (bool success);
        function approve(address _spender, uint256 _value) public returns (bool success);
        function allowance(address _owner, address _spender) public constant returns (uint256 remaining);
        event Transfer(address indexed _from, address indexed _to, uint256 _value);
        event Approval(address indexed _owner, address indexed _spender, uint256 _value);
    }
    contract TokenJCT is Token {
        uint256 public totalSupply;
        string  public name;
        uint8   public decimals;
        string  public symbol;
        address public minter;
       /* This creates an array with all balances */
        mapping (address => uint256) balances;
        mapping (address => mapping (address => uint256)) allowed;
        mapping (address => bool) freezed;
        constructor(uint256 initialAmount, string tokenName, uint8 decimalUnits, string tokenSymbol) public { //knownsec //构造函数
            totalSupply = SafeMath.safeMul(initialAmount , 10 ** uint256(decimalUnits));
            balances[msg.sender] = totalSupply;
            emit Transfer(address(0), msg.sender, totalSupply);
            name = tokenName;
            decimals = decimalUnits;
            symbol = tokenSymbol;

            minter = msg.sender;    //knownsec //铸币者，合约所有者

        }
        /* Send tokens */
        function transfer(address _to, uint256 _value) public returns (bool success) {
            require(!freezed[msg.sender]);

            require(_to != address(0)); //knownsec //验证转账地址不能为空地址

            require(_to != msg.sender); //knownsec //  接收者不能是自己

            require(_value >= 0);
            require(balances[msg.sender] >= _value);
```

```
        require(balances[_to] + _value >= balances[_to]); //knownsec //防止向上溢出
        balances[msg.sender] = SafeMath.safeSub(balances[msg.sender], _value);// Subtract from the sender
        balances[_to] = SafeMath.safeAdd(balances[_to], _value);
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
    /* A contract attempts to get the tokens */
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
        require(!freezed[_from]);
        require(_to != address(0));
        require(_to != _from);
        require(_value >= 0);
        require(_value <= balances[_from]);                    // Check if the sender has enough
        require(_value <= allowed[_from][msg.sender]);         // Check allowance
        require(balances[_to] + _value >= balances[_to]);      // Check for overflows
        balances[_from] = SafeMath.safeSub(balances[_from], _value);     // Subtract from the sender
        balances[_to] = SafeMath.safeAdd(balances[_to], _value);         // Add the same to the recipient
        allowed[_from][msg.sender] = SafeMath.safeSub(allowed[_from][msg.sender], _value);
        emit Transfer(_from, _to, _value);
        return true;
    }
    function balanceOf(address _owner) public constant returns (uint256 balance) {
        return balances[_owner];
    }
    /* Allow another contract to spend some tokens in your behalf */
    function approve(address _spender, uint256 _value) public returns (bool success) {
        require((_value == 0) || (allowed[msg.sender][_spender] == 0));
//knownsec //不存在事务顺序依赖风险
        allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
```

```
        return true;
    }
    function allowance(address _owner, address _spender) public constant returns (uint256 remaining) {
        return allowed[_owner][_spender];
    }
    //冻结账户
    function freezenAccount(address _freezen) public returns (bool success) {
        require(msg.sender == minter);    //knownsec //验证权限
        require(_freezen != minter);
        freezed[_freezen] = true;
        return true;
    }
    //解冻被冻结的账户
    function unFreezenAccount(address _freezen) public returns (bool success) {
        require(msg.sender == minter);
        freezed[_freezen] = false;
        return true;
    }
    //查询账户冻结状态
    function getFrozenState(address _target) public constant returns (bool) {
        return freezed[_target];
    }
    function changeMinter(address _newMinter) public returns (bool) {
        require(msg.sender == minter);
        require(_newMinter != address(0));    //knownsec //防止合约所有者转丢
        minter = _newMinter;
        return true;
    }
   /* only minter can kill */
    function    kill() public{
        require(msg.sender == minter);
        selfdestruct(minter);
    }
}
```

# 5. 附录 B：漏洞风险评级标准

| 智能合约漏洞评级标准 | |
|---|---|
| **漏洞评级** | **漏洞评级说明** |
| **高危漏洞** | 能直接造成代币合约或用户资金损失的漏洞，如：能造成代币价值归零的数值溢出漏洞、能造成交易所损失代币的假充值漏洞、能造成合约账户损失ETH或代币的重入漏洞等；能造成代币合约归属权丢失的漏洞，如：关键函数的访问控制缺陷、call 注入导致关键函数访问控制绕过等；能造成代币合约无法正常工作的漏洞，如：因向恶意地址发送ETH导致的拒绝服务漏洞、因 gas 耗尽导致的拒绝服务漏洞。 |
| **中危漏洞** | 需要特定地址才能触发的高风险漏洞，如代币合约拥有者才能触发的数值溢出漏洞等；非关键函数的访问控制缺陷、不能造成直接资金损失的逻辑设计缺陷等 |
| **低危漏洞** | 难以被触发的漏洞、触发之后危害有限的漏洞，如需要大量ETH或代币才能触发的数值溢出漏洞、触发数值溢出后攻击者无法直接获利的漏洞、通过指定高 gas 触发的事务顺序依赖风险等 |

# 6. 附录 C：漏洞测试工具简介

## 6.1. Manticore

Manticore 是一个分析二进制文件和智能合约的符号执行工具，Manticore 包含一个符号以太坊虚拟机（EVM），一个 EVM 反汇编器/汇编器以及一个用于自动编译和分析 Solidity 的方便界面。它还集成了 Ethersplay，用于 EVM 字节码的 Bit of Traits of Bits 可视化反汇编程序，用于可视化分析。 与二进制文件一样，Manticore 提供了一个简单的命令行界面和一个用于分析 EVM 字节码的 Python API。

## 6.2. Oyente

Oyente 是一个智能合约分析工具，Oyente 可以用来检测智能合约中常见的 bug，比如 reentrancy、事务排序依赖等等。更方便的是，Oyente 的设计是模块化的，所以这让高级用户可以实现并插入他们自己的检测逻辑，以检查他们的合约中自定义的属性。

## 6.3. securify.sh

Securify 可以验证以太坊智能合约常见的安全问题，例如交易乱序和缺少输入验证，它在全自动化的同时分析程序所有可能的执行路径，此外，Securify 还具有用于指定漏洞的特定语言，这使 Securify 能够随时关注当前的安全性和其他可靠性问题。

## 6.4. Echidna

Echidna 是一个为了对 EVM 代码进行模糊测试而设计的 Haskell 库。

## 6.5. MAIAN

MAIAN 是一个用于查找以太坊智能合约漏洞的自动化工具，Maian 处合理约的字节码，并尝试建立一系列交易以找出并确认错误。

## 6.6. ethersplay

ethersplay 是一个 EVM 反汇编器，其中包含了相关分析工具。

## 6.7. ida-evm

ida-evm 是一个针对以太坊虚拟机（EVM）的 IDA 处理器模块。

## 6.8. Remix-ide

Remix 是一款基于浏览器的编译器和 IDE，可让用户使用 Solidity 语言构建以太坊合约并调试交易。

## 6.9. 知道创宇渗透测试人员专用工具包

知道创宇渗透测试人员专用工具包，由知道创宇渗透测试工程师研发，收集和使用，包含专用于测试人员的批量自动测试工具，自主研发的工具、脚本或利用工具等。

知道创宇

北京知道创宇信息技术股份有限公司


咨询电话 +86(10)400 060 9587

邮　　箱 sec@knownsec.com

官　　网 www.knownsec.com

地　　址 北京市 朝阳区 望京 SOHO T2-B座-2509