知道创宇

密级 ★ | 项目组公开

智能合约安全审计报告

审计结果

通过

★ ★ ★ ★ ★

版本说明

修订人	修订内容	修订时间	版本号	审阅人
罗逸锋	编写文档	2020/11/10	V1.0	徐昊杰

文档信息

文档名称	审计日期	审计结果	保密级别	审计查询电话
contracts_ethereum 智能合约安全审计报告	2020/11/10	通过	项目组公开	400-060-9587

版权声明

公司声明

北京知道创宇信息技术股份有限公司基于项目方截至本报告出具时向我方提供的文件和资料，仅对该项目的安全情况进行约定内的安全审计并出具了本报告，我方无法对该项目的背景、项目的实用性、商业模式的合规性、以及项目的合法性等其他情况进行风险判断，亦不对此承担责任。该报告仅供项目方内部决策参考使用，未经我方书面同意，不得擅自将报告予以公开或者提供给其他人或者用于其他目的，我方出具的报告不得作为第三方的任何行为和决策的依据，我方不对用户及第三方因报告采取的相关决策产生的后果承担任何责任。我方假设：截至本报告出具时项目方向我方已提供资料不存在缺失、被篡改、删减或隐瞒的情形，如已提供资料信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符的，我方对由此而导致的损失和不利影响不承担任何责任。

目录

1. 综述

本次报告有效测试时间是从 2020 年 11 月 8 日开始到 2020 年 11 月 10 结束，在此期间针对 contracts_ethereum 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，未发现中、高危安全风险，故综合评定为通过。

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：

项目名称	项目内容
Token 名称	contracts_ethereum
代码类型	代币代码
代码语言	Solidity
代码地址	https://github.com/bitcheck/contracts_ethereum

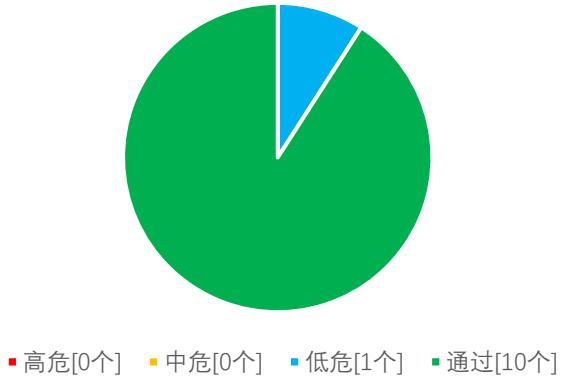
2. 代码漏洞分析

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	1	10

风险等级分布图



2.2. 审计结果汇总

（其他未知安全漏洞不包含在本次审计责任范围）

审计结果			
测试项目	测试内容	状态	描述
智能合约安全审计	重入攻击检测	通过	检查 call.value()函数使用安全
	数值溢出检测	通过	检查 add 和 sub 函数使用安全
	访问控制缺陷检测	通过	检查各操作访问权限控制
	未验证返回值的调用	通过	检查转币方法看是否验证返回值
	错误使用随机数检测	通过	检查是否具备统一的内容过滤器
	事务顺序依赖检测	通过	检查是否存在事务顺序依赖风险
	拒绝服务攻击检测	通过	检查代码在使用资源时是否存在资源滥用问题
	逻辑设计缺陷检测	通过	检查智能合约代码中与业务设计相关的安全问题
	假充值漏洞检测	通过	检查智能合约代码中是否存在假充值漏洞
	增发代币漏洞检测	低危	检查智能合约中是否存在增发代币的功能
	冻结账户绕过检测	通过	检查转移代币中是否存在未校验冻结账户的问题

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区块链智能合约漏洞，曾导致了以太坊的分叉（The DAO hack）。

Solidity 中的 call.value()函数在被用来发送代币的时候会消耗它接收到的所有 gas，当调用 call.value()函数发送代币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相关漏洞。

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代币

，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，可能由于代币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相关漏洞。

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，即这些随机数载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相关漏洞。

安全建议：无。

3.7. 拒绝服务攻击【通过】

在区块链的世界中，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相关漏洞。

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相关漏洞。

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(ABBT.sender)的余额检查用的是 if 判断方式，当 balances[ABBT.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相关漏洞。

安全建议：无。

3.10. 增发代币漏洞【低危】

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，智能合约代码中存在相关漏洞。

contracts/Mocks/BTCHToken.sol

```
41      function mint(address account, uint256 amount) public onlyAuthorizedContract {
42          if(_totalSupply.add(amount) > maxSupply) amount = _totalSupply.add(amount).sub(maxSupply);
43          if(amount > 0) _mint(account, amount);
44      }
```

安全建议：该问题不属于安全问题，但部分交易所会限制增发函数的使用，具体情况需根据交易所的要求而定。

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问题。

安全建议：无。

4. 附录 A：合约代码

DividendPool.sol

```
/**
 *  $$$$$$\  $$\                $$\
 * $$  __$$\ $$ |               $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ---------------------------------------------------------------------
*/

pragma solidity >=0.4.23 <0.6.0;

import "./Mocks/BTCHToken.sol";
import "./ReentrancyGuard.sol";
import "./Mocks/SafeMath.sol";

contract DividendPool is ReentrancyGuard {
    using SafeMath for uint256;
    uint256 public totalBalance = 0;
    address public tokenAddress;    // BTCH
    address public dividentAddress; // USDT Token
    address public feeAddress; // must be same as commonWithdrawAddress in ShakerV2.sol
    address public operator;

    // Share dividents of fee
    uint256 public currentStartTimestamp = 0;
    uint256 public totalDividents = 0;
    uint256 public sentDividents = 0;
    uint256 public getDividentsTimeout = 172800;// Have 2 days to getting current
dividents

    event Dividend(address to, uint256 amount, uint256 timestamp);

    mapping(address => uint256) private lastGettingDividentsTime;
    mapping(address => uint256) public balances;

    BTCHToken public token;
    ERC20 public dividentToken;

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    constructor(
      address _tokenAddress,
      address _dividentAddress,
      address _feeAddress
    ) public {
        tokenAddress = _tokenAddress;
        token = BTCHToken(tokenAddress);
        dividentAddress = _dividentAddress;
        dividentToken = ERC20(dividentAddress);
        operator = msg.sender;
        feeAddress = _feeAddress;
    }

    function depositBTCH(uint256 amount) external nonReentrant {
        require(amount > 0);
        require(!(block.timestamp <= currentStartTimestamp + getDividentsTimeout)
|| !(block.timestamp >= currentStartTimestamp), "You can not deposit during taking
divident time");
        require(amount <= token.balanceOf(msg.sender), "Your balance is not enough");
        require(token.allowance(msg.sender, address(this)) >= amount, "Your allowance
is not enough");
        token.transferFrom(msg.sender, address(this), amount);
        balances[msg.sender] = balances[msg.sender].add(amount);
```

```
        totalBalance = totalBalance.add(amount);
    }

    function withdrawBTCH(uint256 amount) external nonReentrant {
        require(amount > 0);
        require(!(block.timestamp <= currentStartTimestamp + getDividentsTimeout)
|| !(block.timestamp >= currentStartTimestamp), "You can not withdraw during taking
divident time");
        require(amount <= balances[msg.sender], "Your deposit balance is not enough");
        token.transfer(msg.sender, amount);
        balances[msg.sender] = balances[msg.sender].sub(amount);
        totalBalance = totalBalance.sub(amount);
    }

    function updateTokenAddress(address _addr) external onlyOperator nonReentrant {
      require(_addr != address(0));
      tokenAddress = _addr;
      token = BTCHToken(tokenAddress);
    }

    function updateOperator(address _addr) external onlyOperator nonReentrant {
      require(_addr != address(0));
      operator = _addr;
    }

    function getBalance() external view returns(uint256) {
      return balances[msg.sender];
    }

    function getDividentsAmount() public view returns(uint256, uint256) {
      // Caculate normal dividents
      require(totalBalance > 0);
      return (totalDividents.mul(balances[msg.sender]).div(totalBalance),
lastGettingDividentsTime[msg.sender]);
    }

    function setDividentAddress(address _address) external onlyOperator {
      dividentAddress = _address;
      dividentToken = ERC20(dividentAddress);
    }

    function setFeeAddress(address _address) external onlyOperator {
        require(_address != address(0));
        feeAddress = _address;
    }

    function sendDividents() external nonReentrant {
      // Only shaker contract can call this function
      require(block.timestamp <= currentStartTimestamp + getDividentsTimeout &&
block.timestamp >= currentStartTimestamp, "Getting dividents not start or it's already
end");
      require(lastGettingDividentsTime[msg.sender] < currentStartTimestamp, "You have
got dividents already");
      (uint256 normalDividents,) = getDividentsAmount();

      // Send Dividents
      // The fee account must approve the this contract enough allowance of USDT as
dividend
      require(dividentToken.allowance(feeAddress, address(this)) >= normalDividents,
"Allowance not enough");
      dividentToken.transferFrom(feeAddress, msg.sender, normalDividents);
      sentDividents = sentDividents.add(normalDividents);
      lastGettingDividentsTime[msg.sender] = block.timestamp;
      emit Dividend(msg.sender, normalDividents, block.timestamp);
    }

    /** Start Dividents by operator */
    function startDividents(uint256 from, uint256 amount) external onlyOperator
nonReentrant{
      require(from > block.timestamp);
      require(amount > 0);
      currentStartTimestamp = from;
      totalDividents = amount;
      sentDividents = 0;
    }

    function setGettingDividentsTimeout(uint256 _seconds) external onlyOperator {
```

```
    getDividentsTimeout = _seconds;
  }

  function getLastTakingDividentsTime() external view returns(uint256) {
    return lastGettingDividentsTime[msg.sender];
  }
}
```

ERC20ShakerV2.sol

```
/**
 *  $$$$$$\  $$\                $$\
 * $$  __$$\ $$ |               $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ----------------------------------------------------------------------
*/

pragma solidity >=0.4.23 <0.6.0;

import "./ShakerV2.sol";

contract ERC20ShakerV2 is ShakerV2 {
  address public token;

  constructor(
    address _operator,
    address _commonWithdrawAddress,
    address _token
  ) ShakerV2(_operator, _commonWithdrawAddress) public {
    token = _token;
  }

  function _processDeposit(uint256 _amount) internal {
    require(msg.value == 0, "ETH value is supposed to be 0 for ERC20 instance");
    _safeErc20TransferFrom(msg.sender, address(this), _amount);
  }

  function _processWithdraw(address payable _recipient, address _relayer, uint256
_fee, uint256 _refund) internal {
    _safeErc20Transfer(_recipient, _refund.sub(_fee));
    if(_fee > 0) _safeErc20Transfer(_relayer, _fee);
  }

  function _safeErc20TransferFrom(address _from, address _to, uint256 _amount)
internal {
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd /*
transferFrom */, _from, _to, _amount));
    require(success, "not enough allowed tokens");

    // if contract returns some data lets make sure that is `true` according to
standard
    if (data.length > 0) {
      require(data.length == 32, "data length should be either 0 or 32 bytes");
      success = abi.decode(data, (bool));
      require(success, "not enough allowed tokens. Token returns false.");
    }
  }

  function _safeErc20Transfer(address _to, uint256 _amount) internal {
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0xa9059cbb /*
transfer */, _to, _amount));
    require(success, "not enough tokens");

    // if contract returns some data lets make sure that is `true` according to
standard
    if (data.length > 0) {
      require(data.length == 32, "data length should be either 0 or 32 bytes");
      success = abi.decode(data, (bool));
      require(success, "not enough tokens. Token returns false.");
    }
```

```
  }
}
```

ShakerTokenManager.sol

```
/**
 *  $$$$$$\  $$\                $$\
 * $$  __$$\ $$ |               $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ---------------------------------------------------------
*/

pragma solidity >=0.4.23 <0.6.0;

import "./Mocks/BTCHToken.sol";
import "./Mocks/ERC20.sol";
import "./ReentrancyGuard.sol";

/**
 * The bonus will calculated with 5 factors:
 * 1- Base bonus factor. This is base bonus factor, here we set it as 0.05
 * 2- Amount after exponent. This will reduce the weight of whale capital. Here we set
it as 2/3. That means if somebody deposit 100,000, will just caculatd busnus
 *   according to 100,000 ** (2/3) = 2154.435
 * 3- Hours factor between Deposit Withdraw. This is let the people store the money in
contract longer.
 *   If below 1 hour, there will be no bonus token.
 *   from 1-24 hours, the factor is 0.05; 24-48 hours, factor is 0.15, etc.
 *   This factor will be modified by council according to the market.
 * 4- Stage factor: If total mint token is 300,000, we will devided it into several
stages. Each stage has special bonus times. Ex. if stage factor is 5,
 *   means in this stage, miner will get 5 times than normal.
 * 5- Price elastical factor. We want the mint quantity for each deposit can be
different under different market. If the price is higher than normal, the factor will
 *   become smaller automaticly, and if the price go down, the factor will become
smaller also. It is a Gaussian distribution, and the average price (normal price)
 *   is fee of deposit and withdrawal.
 *
 * So the bonus amount will be:
 * Bonus amount = Amount after exponent * Base bonus factor * hours factor * stage
factor * price elastical factor.
 *
 * In this version, we will keep price elastical factor as 1.
 *
 */
contract ShakerTokenManager is ReentrancyGuard {
    using SafeMath for uint256;

    uint256 public bonusTokenDecimals = 6; // bonus token decimals, BTCH
    uint256 public depositTokenDecimals = 6; // deposit and withdrawal token decimals,
USDT

    // Params
    uint256 public baseFactor = 50; // 50 means 0.05
    uint256[] public intervalOfDepositWithdraw = [1, 24, 48, 96, 192, 384, 720]; //
hours of inverval between deposit and withdraw
    uint256[] public intervalOfDepositWithdrawFactor = [5000, 15000, 16800, 20600,
28600, 45500, 81500]; // 5000 will be devided by 1e5, means 0.05
    uint256[] public stageFactors = [10000, 5000, 2500, 1250, 625]; // Stage factor,
5000 means 5, 2500 means 2.5, etc.
    uint256 public eachStageAmount = 7200000 * 10 ** bonusTokenDecimals; // Each stage
amount
    uint256[] public exponent = [2, 3];// means 2/3
    uint256 public feeRate = 33333; // 33.333% will be 33333
    uint256 public minChargeFeeAmount = 10 * 10 ** depositTokenDecimals;// Below this
amount, will only charge  very special fee
    uint256 public minChargeFee = 0; // min amount of special charge.
    uint256 public minChargeFeeRate = 180; // percent rate of special charge, if need
to charge 1.5%, this will be set 150
    uint256 public minMintAmount = 10 * 10 ** depositTokenDecimals;
    uint256 public taxRate = 2000;// means 20%
```

```
    uint256 public depositerShareRate = 5000; // depositer and withdrawer will share
the bonus, this rate is for sender(depositer). 5000 means 0.500, 50%;

    address public operator;
    address public taxBereauAddress; // address to get tax
    address public shakerContractAddress;
    address public tokenAddress; // BTCH token

    BTCHToken public token = BTCHToken(tokenAddress);

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    modifier onlyShaker {
        require(msg.sender == shakerContractAddress, "Only bitcheck contract can call
this function.");
        _;
    }

    constructor(address _shakerContractAddress, address _taxBereauAddress) public {
        operator = msg.sender;
        shakerContractAddress = _shakerContractAddress;
        taxBereauAddress = _taxBereauAddress;
    }

    function sendBonus(uint256 _amount, uint256 _hours, address _depositer, address
_withdrawer) external nonReentrant onlyShaker returns(bool) {
        uint256 mintAmount = this.getMintAmount(_amount, _hours);
        uint256 tax = mintAmount.mul(taxRate).div(10000);
        uint256 notax = mintAmount.sub(tax);
        token.mint(_depositer, (notax.mul(depositerShareRate).div(10000)));
        token.mint(_withdrawer,
(notax.mul(uint256(10000).sub(depositerShareRate)).div(10000)));
        token.mint(taxBereauAddress, tax);
        return true;
    }

    function burn(uint256 _amount, address _from) external nonReentrant onlyShaker
returns(bool) {
        token.burn(_from, _amount);
        return true;
    }

    function getMintAmount(uint256 _amount, uint256 _hours ) external view
returns(uint256) {
        // return back bonus token amount with decimals
        require(_amount < 1e18);
        if(_amount <= minMintAmount) return 0;
        uint256 amountExponented = getExponent(_amount);
        uint256 stageFactor = getStageFactor();
        uint256 intervalFactor = getIntervalFactor(_hours);
        uint256 priceFactor = getPriceElasticFactor();
        return
amountExponented.mul(priceFactor).mul(baseFactor).mul(intervalFactor).mul(stageFactor)
.div(1e11);
    }

    function getFee(uint256 _amount) external view returns(uint256) {
        // return fee amount, including decimals
        require(_amount < 1e18);
        if(_amount <= minChargeFeeAmount) return getSpecialFee(_amount);
        uint256 amountExponented = getExponent(_amount);
        return amountExponented.mul(feeRate).div(1e5);
    }

    function getExponent(uint256 _amount) internal view returns(uint256) {
        // if 2000, the _amount should be 2000 * 10**decimals, return back 2000**(2/3)
* 10**decimals
        if(_amount > 1e18) return 0;
        uint256 e = nthRoot(_amount, exponent[1], bonusTokenDecimals, 1e18);
        return e.mul(e).div(10 ** (bonusTokenDecimals + depositTokenDecimals *
exponent[0] / exponent[1]));
    }

    function getStageFactor() internal view returns(uint256) {
```

```
        uint256 tokenTotalSupply = getTokenTotalSupply();
        uint256 stage = tokenTotalSupply.div(eachStageAmount);
        return stageFactors[stage > stageFactors.length - 1 ? stageFactors.length - 1 :
stage];
    }

    function getIntervalFactor(uint256 _hours) internal view returns(uint256) {
        uint256 id = intervalOfDepositWithdraw.length - 1;
        for(uint8 i = 0; i < intervalOfDepositWithdraw.length; i++) {
            if(intervalOfDepositWithdraw[i] > _hours) {
                id = i == 0 ? 999 : i - 1;
                break;
            }
        }
        return id == 999 ? 0 : intervalOfDepositWithdrawFactor[id];
    }

    // For tesing, Later will update ######
    function getPriceElasticFactor() internal pure returns(uint256) {
        return 1;
    }

    function getTokenTotalSupply() public view returns(uint256) {
        return token.totalSupply();
    }

    function getSpecialFee(uint256 _amount) internal view returns(uint256) {
        return _amount.mul(minChargeFeeRate).div(10000).add(minChargeFee);
    }
    // calculates a^(1/n) to dp decimal places
    // maxIts bounds the number of iterations performed
    function nthRoot(uint _a, uint _n, uint _dp, uint _maxIts) internal pure
returns(uint) {
        assert (_n > 1);

        // The scale factor is a crude way to turn everything into integer calcs.
        // Actually do (a * (10 ^ ((dp + 1) * n))) ^ (1/n)
        // We calculate to one extra dp and round at the end
        uint one = 10 ** (1 + _dp);
        uint a0 = one ** _n * _a;

        // Initial guess: 1.0
        uint xNew = one;
        uint x;
        uint iter = 0;
        while (xNew != x && iter < _maxIts) {
            x = xNew;
            uint t0 = x ** (_n - 1);
            if (x * t0 > a0) {
                xNew = x - (x - a0 / t0) / _n;
            } else {
                xNew = x + (a0 / t0 - x) / _n;
            }
            ++iter;
        }

        // Round to nearest in the last dp.
        return (xNew + 5) / 10;
    }

    function setStageFactors(uint256[] calldata _stageFactors) external onlyOperator {
        stageFactors = _stageFactors;
    }

    function setIntervalOfDepositWithdraw(uint256[] calldata
_intervalOfDepositWithdraw, uint256[] calldata _intervalOfDepositWithdrawFactor)
external onlyOperator {
      intervalOfDepositWithdrawFactor = _intervalOfDepositWithdrawFactor;
      intervalOfDepositWithdraw = _intervalOfDepositWithdraw;
    }

    function setBaseFactor(uint256 _baseFactor) external onlyOperator {
        baseFactor = _baseFactor;
    }

    function setBonusTokenDecimals(uint256 _decimals) external onlyOperator {
        require(_decimals >= 0);
```

```
        bonusTokenDecimals = _decimals;
    }

    function setDeositTokenDecimals(uint256 _decimals) external onlyOperator {
        require(_decimals >= 0);
        depositTokenDecimals = _decimals;
    }

    function setTokenAddress(address _address) external onlyOperator {
        tokenAddress = _address;
        token = BTCHToken(tokenAddress);
    }

    function setShakerContractAddress(address _shakerContractAddress) external
onlyOperator {
        shakerContractAddress = _shakerContractAddress;
    }

    function setExponent(uint256[] calldata _exp) external onlyOperator {
        require(_exp.length == 2 && _exp[1] >= 0);
        exponent = _exp;
    }

    function setEachStageAmount(uint256 _eachStageAmount) external onlyOperator {
        require(_eachStageAmount >= 0);
        eachStageAmount = _eachStageAmount;
    }

    function setMinChargeFeeParams(uint256 _maxAmount, uint256 _minFee, uint256
_feeRate) external onlyOperator {
        minChargeFeeAmount = _maxAmount;
        minChargeFee = _minFee;
        minChargeFeeRate = _feeRate;
    }

    function setMinMintAmount(uint256 _amount) external onlyOperator {
        require(_amount >= 0);
        minMintAmount = _amount;
    }

    function setFeeRate(uint256 _feeRate) external onlyOperator {
        require(_feeRate <= 100000 && _feeRate >= 0);
        feeRate = _feeRate;
    }

    function setTaxBereauAddress(address _taxBereauAddress) external onlyOperator {
        taxBereauAddress = _taxBereauAddress;
    }

    function setTaxRate(uint256 _rate) external onlyOperator {
        require(_rate <= 10000 && _rate >= 0);
        taxRate = _rate;
    }

    function setDepositerShareRate(uint256 _rate) external onlyOperator {
        require(_rate <= 10000 && _rate >= 0);
        depositerShareRate = _rate;
    }

    function updateOperator(address _newOperator) external onlyOperator {
        require(_newOperator != address(0));
        operator = _newOperator;
    }

}
```

ShakerV2.sol

```
/**
 *  $$$$$$\  $$\                 $$\
 * $$  __$$\ $$ |                $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
```

```
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 * ______________________________________________________________________
*/

pragma solidity >=0.4.23 <0.6.0;

import "./ReentrancyGuard.sol";
import "./StringUtils.sol";
import "./ShakerTokenManager.sol";

contract ShakerV2 is ReentrancyGuard, StringUtils {
    using SafeMath for uint256;
    uint256 public totalAmount = 0; // Total amount of deposit
    uint256 public totalBalance = 0; // Total balance of deposit after Withdrawal

    address public operator;          // Super operator account to control the contract
    address public councilAddress;     // Council address of DAO
    uint256 public councilJudgementFee = 0; // Council charge for judgement
    uint256 public councilJudgementFeeRate = 1700; // If the desired rate is 17%,
commonFeeRate should set to 1700

    ShakerTokenManager public tokenManager;

    struct Commitment {              // Deposit Commitment
        bool          status;           // If there is no this commitment or balance is
zeor, false
        uint256       amount;           // Deposit balance
        address payable sender;        // Who make this deposit
        uint256       effectiveTime;     // Forward cheque time
        uint256       timestamp;        // Deposit timestamp
        bool          canEndorse;
        bool          lockable;         // If can be locked/refund
    }
    // Mapping of commitments, must be private. The key is hashKey =
hash(commitment,recipient)
    // The contract will hide the recipient and commitment while make deposit.
    mapping(bytes32 => Commitment) private commitments;

    // Relayer is service to do the deposit and Withdrawal on server, this address is
for recieving fee
    mapping(address => address) private relayerWithdrawAddress;

    // If the msg.sender(relayer) has not registered Withdrawal address, the fee will
send to this address
    address public commonWithdrawAddress;

    // If withdrawal is not throught relayer, use this common fee. Be care of decimal
of token
    // uint256 public commonFee = 0;

    // If withdrawal is not throught relayer, use this rate. Total fee is: commoneFee +
amount * commonFeeRate.
    // If the desired rate is 4%, commonFeeRate should set to 400
    // uint256 public commonFeeRate = 25; // 0.25%

    struct LockReason {
        string  description;
        uint8   status; // 1- locked, 2- confirm by recipient, 0- never happend, 3-
unlocked by council, 4- cancel by sender
        uint256 datetime;
        uint256 refund;
        address payable locker;
        bool    recipientAgree;
        bool    senderAgree;
        bool    toCouncil;
    }
    // locakReason key is hashKey = hash(commitment, recipient)
    mapping(bytes32 => LockReason) private lockReason;

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    modifier onlyRelayer {
        require(relayerWithdrawAddress[msg.sender] != address(0x0), "Only relayer can
call this function.");
```

```
        _;
    }

    modifier onlyCouncil {
        require(msg.sender == councilAddress, "Only council account can call this
function.");
        _;
    }

    event Deposit(address sender, bytes32 hashkey, uint256 amount, uint256 timestamp);
    event Withdrawal(string commitment, uint256 fee, uint256 amount, uint256
timestamp);

    constructor(
        address _operator,
        address _commonWithdrawAddress
    ) public {
        operator = _operator;
        councilAddress = _operator;
        commonWithdrawAddress = _commonWithdrawAddress;
    }

    function depositERC20Batch(
        bytes32[] calldata _hashKey,
        uint256[] calldata _amounts,
        uint256[] calldata _effectiveTime
    ) external payable nonReentrant {
        for(uint256 i = 0; i < _amounts.length; i++) {
            _deposit(_hashKey[i], _amounts[i], _effectiveTime[i]);
        }
    }

    function _deposit(
        bytes32 _hashKey,
        uint256 _amount,
        uint256 _effectiveTime
    ) internal {
        require(!commitments[_hashKey].status, "The commitment has been submitted or
used out.");
        require(_amount > 0);

        _processDeposit(_amount);

        commitments[_hashKey].status = true;
        commitments[_hashKey].amount = _amount;
        commitments[_hashKey].sender = msg.sender;
        commitments[_hashKey].effectiveTime = _effectiveTime < block.timestamp ?
block.timestamp : _effectiveTime;
        commitments[_hashKey].timestamp = block.timestamp;
        commitments[_hashKey].canEndorse = false;
        commitments[_hashKey].lockable = true;

        totalAmount = totalAmount.add(_amount);
        totalBalance = totalBalance.add(_amount);

        emit Deposit(msg.sender, _hashKey, _amount, block.timestamp);
    }

    function _processDeposit(uint256 _amount) internal;

    function withdrawERC20Batch(
        bytes32[] calldata _commitments,
        uint256[] calldata _amounts,
        uint256[] calldata _fees,
        address[] calldata _relayers
    ) external payable nonReentrant {
        for(uint256 i = 0; i < _commitments.length; i++)
_withdraw(bytes32ToString(_commitments[i]), _amounts[i], _fees[i], _relayers[i]);
    }

    function _withdraw(
        string memory _commitment,
        uint256 _amount,             // Withdrawal amount
        uint256 _fee,                // Fee caculated by relayer
        address _relayer             // Relayer address
    ) internal {
        bytes32 _hashkey = getHashkey(_commitment);
```

```
        require(commitments[_hashkey].amount > 0, 'The commitment of this recipient is
not exist or used out');
        require(lockReason[_hashkey].status != 1, 'This deposit was locked');
        uint256 refundAmount = _amount < commitments[_hashkey].amount ? _amount :
commitments[_hashkey].amount; //Take all if _refund == 0
        require(refundAmount > 0, "Refund amount can not be zero");
        require(block.timestamp >= commitments[_hashkey].effectiveTime, "The deposit is
locked until the effectiveTime");
        require(refundAmount >= _fee, "Refund amount should be more than fee");

        address relayer = relayerWithdrawAddress[_relayer] == address(0x0) ?
commonWithdrawAddress : relayerWithdrawAddress[_relayer];
        uint256 _fee1 = tokenManager.getFee(refundAmount);
        require(_fee1 <= refundAmount, "The fee can not be more than refund amount");
        uint256 _fee2 = relayerWithdrawAddress[_relayer] == address(0x0) ? _fee1 :
_fee; // If not through relay, use commonFee
        _processWithdraw(msg.sender, relayer, _fee2, refundAmount);

        commitments[_hashkey].amount =
(commitments[_hashkey].amount).sub(refundAmount);
        commitments[_hashkey].status = commitments[_hashkey].amount <= 0 ? false :
true;
        totalBalance = totalBalance.sub(refundAmount);

        uint256 _hours =
(block.timestamp.sub(commitments[_hashkey].timestamp)).div(3600);
        tokenManager.sendBonus(refundAmount, _hours, commitments[_hashkey].sender,
msg.sender);

        emit Withdrawal(_commitment, _fee, refundAmount, block.timestamp);
    }

    function _processWithdraw(address payable _recipient, address _relayer, uint256
_fee, uint256 _refund) internal;
    function _safeErc20Transfer(address _to, uint256 _amount) internal;

    function getHashkey(string memory _commitment) internal view returns(bytes32) {
        string memory commitAndTo = concat(_commitment, addressToString(msg.sender));
        return keccak256(abi.encodePacked(commitAndTo));
    }

    function endorseERC20Batch(
        uint256[] calldata _amounts,
        bytes32[] calldata _oldCommitments,
        bytes32[] calldata _newHashKeys,
        uint256[] calldata _effectiveTimes
    ) external payable nonReentrant {
        for(uint256 i = 0; i < _amounts.length; i++) _endorse(_amounts[i],
bytes32ToString(_oldCommitments[i]), _newHashKeys[i], _effectiveTimes[i]);
    }

    function _endorse(
        uint256 _amount,
        string memory _oldCommitment,
        bytes32 _newHashKey,
        uint256 _effectiveTime
    ) internal {
        bytes32 _oldHashKey = getHashkey(_oldCommitment);
        require(lockReason[_oldHashKey].status != 1, 'This deposit was locked');
        require(commitments[_oldHashKey].status, "Old commitment can not find");
        require(!commitments[_newHashKey].status, "The new commitment has been
submitted or used out");
        require(commitments[_oldHashKey].canEndorse, "Old commitment can not endorse");
        require(commitments[_oldHashKey].amount > 0, "No balance amount of this
proof");
        uint256 refundAmount = _amount < commitments[_oldHashKey].amount ? _amount :
commitments[_oldHashKey].amount; //Take all if _refund == 0
        require(refundAmount > 0, "Refund amount can not be zero");

        if(_effectiveTime > 0 && block.timestamp >=
commitments[_oldHashKey].effectiveTime) commitments[_oldHashKey].effectiveTime =
_effectiveTime; // Effective
        else commitments[_newHashKey].effectiveTime =
commitments[_oldHashKey].effectiveTime; // Not effective

        commitments[_newHashKey].status = true;
        commitments[_newHashKey].amount = refundAmount;
```

```
        commitments[_newHashKey].sender = msg.sender;
        commitments[_newHashKey].timestamp = block.timestamp;
        commitments[_newHashKey].canEndorse = false;
        commitments[_newHashKey].lockable = true;

        commitments[_oldHashKey].amount =
(commitments[_oldHashKey].amount).sub(refundAmount);
        commitments[_oldHashKey].status = commitments[_oldHashKey].amount <= 0 ?
false : true;

        emit Withdrawal(_oldCommitment,  0, refundAmount, block.timestamp);
        emit Deposit(msg.sender, _newHashKey, refundAmount, block.timestamp);
    }

    /** @dev whether a note is already spent */
    function isSpent(bytes32 _hashkey) public view returns(bool) {
        return commitments[_hashkey].amount == 0 ? true : false;
    }

    /** @dev whether an array of notes is already spent */
    function isSpentArray(bytes32[] calldata _hashkeys) external view returns(bool[]
memory spent) {
        spent = new bool[](_hashkeys.length);
        for(uint i = 0; i < _hashkeys.length; i++) spent[i] = isSpent(_hashkeys[i]);
    }

    /** @dev operator can change his address */
    function updateOperator(address _newOperator) external nonReentrant onlyOperator {
        operator = _newOperator;
    }

    /** @dev update authority relayer */
    function updateRelayer(address _relayer, address _withdrawAddress) external
nonReentrant onlyOperator {
        relayerWithdrawAddress[_relayer] = _withdrawAddress;
    }

    /** @dev get relayer Withdrawal address */
    function getRelayerWithdrawAddress() view external onlyRelayer returns(address) {
        return relayerWithdrawAddress[msg.sender];
    }

    /** @dev update commonWithdrawAddress */
    function updateCommonWithdrawAddress(address _commonWithdrawAddress) external
nonReentrant onlyOperator {
        commonWithdrawAddress = _commonWithdrawAddress;
    }

    /** @dev set council address */
    function setCouncial(address _councilAddress) external nonReentrant onlyOperator {
        councilAddress = _councilAddress;
    }

    /** @dev lock commitment, this operation can be only called by note holder */
    function lockERC20Batch (
        bytes32           _hashkey,
        uint256           _refund,
        string  calldata  _description
    ) external payable nonReentrant {
        _lock(_hashkey, _refund, _description);
    }

    function _lock(
        bytes32 _hashkey,
        uint256 _refund,
        string memory _description
    ) internal {
        require(msg.sender == commitments[_hashkey].sender, 'Locker must be sender');
        require(commitments[_hashkey].lockable, 'This commitment must be lockable');
        require(commitments[_hashkey].amount >= _refund, 'Balance amount must be
enough');

        lockReason[_hashkey] = LockReason(
            _description,
            1,
            block.timestamp,
            _refund == 0 ? commitments[_hashkey].amount : _refund,
```

```
            msg.sender,
            false,
            false,
            false
        );
    }

    function getLockReason(bytes32 _hashkey) public view returns(
        string memory  description,
        uint8          status,
        uint256        datetime,
        uint256        refund,
        address        locker,
        bool           recipientAgree,
        bool           senderAgree,
        bool           toCouncil
    ) {
        LockReason memory data = lockReason[_hashkey];
        return (
            data.description,
            data.status,
            data.datetime,
            data.refund,
            data.locker,
            data.recipientAgree,
            data.senderAgree,
            data.toCouncil
        );

    }

    function unlockByCouncil(bytes32 _hashkey, uint8 _result) external nonReentrant
onlyCouncil {
        // _result = 1: sender win
        // _result = 2: recipient win
        require(_result == 1 || _result == 2);
        if(lockReason[_hashkey].status == 1 && lockReason[_hashkey].toCouncil) {
            lockReason[_hashkey].status = 3;
            // If the council decided to return back money to the sender
            uint256 councilFee = getJudgementFee(lockReason[_hashkey].refund);
            if(_result == 1) {
                _processWithdraw(lockReason[_hashkey].locker, councilAddress, councilFee,
lockReason[_hashkey].refund);
                totalBalance = totalBalance.sub(lockReason[_hashkey].refund);
                commitments[_hashkey].amount =
(commitments[_hashkey].amount).sub(lockReason[_hashkey].refund);
                commitments[_hashkey].status = commitments[_hashkey].amount == 0 ?
false : true;
            } else {
                lockReason[_hashkey].status = 3;
                _safeErc20Transfer(councilAddress, councilFee);
                totalBalance = totalBalance.sub(councilFee);
                commitments[_hashkey].amount =
(commitments[_hashkey].amount).sub(councilFee);
                commitments[_hashkey].status = commitments[_hashkey].amount == 0 ?
false : true;
            }
        }
    }

    /**
     * recipient should agree to let sender refund, otherwise, will bring to the
council to make a judgement
     * This is 1st step if dispute happend
     */
    function unlockByRecipent(bytes32 _hashkey, bytes32 _commitment, uint8 _status)
external nonReentrant {
        bytes32 _recipientHashKey = getHashkey(bytes32ToString(_commitment));
        bool isSender = msg.sender == commitments[_hashkey].sender;
        bool isRecipent = _hashkey == _recipientHashKey;

        require(isSender || isRecipent, 'Must be called by recipient or original
sender');
        require(_status == 1 || _status == 2);
        require(lockReason[_hashkey].status == 1);

        if(isSender) {
```

```
            // Sender accept to keep cheque available
            lockReason[_hashkey].status = _status == 2 ? 4 : 1;
            lockReason[_hashkey].senderAgree = _status == 2;
            lockReason[_hashkey].toCouncil = _status == 2;
        } else if(isRecipent) {
            // recipient accept to refund back to sender
            lockReason[_hashkey].status = _status;
            lockReason[_hashkey].recipientAgree = _status == 2;
            lockReason[_hashkey].toCouncil = _status == 2;
            // return back to sender
            if(_status == 2) {
                _processWithdraw(commitments[_hashkey].sender, address(0x0), 0,
lockReason[_hashkey].refund);
                totalBalance = totalBalance.sub(lockReason[_hashkey].refund);
                commitments[_hashkey].amount =
(commitments[_hashkey].amount).sub(lockReason[_hashkey].refund);
                commitments[_hashkey].status = commitments[_hashkey].amount == 0 ?
false : true;
            } else {
                lockReason[_hashkey].toCouncil = true;
            }
        }
    }

    /**
     * Cancel effectiveTime and change cheque to at sight
     */
    function changeToAtSight(bytes32 _hashkey) external nonReentrant returns(bool) {
        require(msg.sender == commitments[_hashkey].sender, 'Only sender can change
this cheque to at sight');
        if(commitments[_hashkey].effectiveTime > block.timestamp)
commitments[_hashkey].effectiveTime = block.timestamp;
        return true;
    }

    function setCanEndorse(bytes32 _hashkey, bool status) external nonReentrant
returns(bool) {
        require(msg.sender == commitments[_hashkey].sender, 'Only sender can change
endorsable');
        commitments[_hashkey].canEndorse = status;
    }

    function setLockable(bytes32 _hashKey, bool status) external nonReentrant
returns(bool) {
        require(msg.sender == commitments[_hashKey].sender, 'Only sender can change
lockable');
        require(commitments[_hashKey].lockable == true && status == false, 'Can only
change from lockable to non-lockable');
        commitments[_hashKey].lockable = status;
        commitments[_hashKey].canEndorse = true; // If the commitment can not be lock,
it must be endorsed
    }

    function getDepositDataByHashkey(bytes32 _hashkey) external view returns(uint256
effectiveTime, uint256 amount, bool lockable, bool canEndorse) {
        effectiveTime = commitments[_hashkey].effectiveTime;
        amount = commitments[_hashkey].amount;
        lockable = commitments[_hashkey].lockable;
        canEndorse = commitments[_hashkey].canEndorse;
    }

    function updateCouncilJudgementFee(uint256 _fee, uint256 _rate) external
nonReentrant onlyCouncil {
        councilJudgementFee = _fee;
        councilJudgementFeeRate = _rate;
    }

    function updateBonusTokenManager(address _BonusTokenManagerAddress) external
nonReentrant onlyOperator {
        tokenManager = ShakerTokenManager(_BonusTokenManagerAddress);
    }

    function getJudgementFee(uint256 _amount) internal view returns(uint256) {
        return _amount * councilJudgementFeeRate / 10000 + councilJudgementFee;
    }
}
```

Mocks/BTCHToken.sol

```
/**
 *  $$$$$$\  $$\                 $$\
 * $$  __$$\ $$ |                $$ |
 * $$ /  \__|$$$$$$$\   $$$$$$\  $$ |  $$\  $$$$$$\   $$$$$$\
 * \$$$$$$\  $$  __$$\  \____$$\ $$ | $$  |$$  __$$\ $$  __$$\
 *  \____$$\ $$ |  $$ | $$$$$$$ |$$$$$$  / $$$$$$$$ |$$ |  \__|
 * $$\   $$ |$$ |  $$ |$$  __$$ |$$  _$$<  $$   ____|$$ |
 * \$$$$$$  |$$ |  $$ |\$$$$$$$ |$$ | \$$\ \$$$$$$$\ $$ |
 *  \______/ \__|  \__| \_______|\__|  \__| \_______|\__|
 * $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 *
*/

pragma solidity >=0.4.23 <0.6.0;

import "./ERC20.sol";
import "./ERC20Detailed.sol";

contract BTCHToken is ERC20, ERC20Detailed {
    address public authorizedContract;
    address public operator;
    uint256 public maxSupply = 36000000 * 10 ** 6;

    constructor (address _authorizedContract) public ERC20Detailed("BitCheck DAO",
"BTCH", 6) {
        // Decimal is 6
        operator = msg.sender;
        authorizedContract = _authorizedContract;
        // zero pre-mine
    }

    modifier onlyOperator {
        require(msg.sender == operator, "Only operator can call this function.");
        _;
    }

    modifier onlyAuthorizedContract {
        require(msg.sender == authorizedContract, "Only authorized contract can call
this function.");
        _;
    }

    function mint(address account, uint256 amount) public onlyAuthorizedContract {
        if(_totalSupply.add(amount) > maxSupply) amount =
_totalSupply.add(amount).sub(maxSupply);
        if(amount > 0) _mint(account, amount);
    }

    function burn(address account, uint256 amount) public onlyAuthorizedContract {
        if(_totalSupply < amount) amount = amount.sub(_totalSupply);
        if(amount > 0) _burn(account, amount);
    }

    function updateOperator(address _newOperator) external onlyOperator {
        require(_newOperator != address(0));
        operator = _newOperator;
    }

    function updateAuthorizedContract(address _authorizedContract) external
onlyOperator {
        require(_authorizedContract != address(0));
        authorizedContract = _authorizedContract;
    }

}
```

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. MaABBTicore

MaABBTicore 是一个分析二进制文件和智能合约的符号执行工具，MaABBTicore 包含一个符号区块链虚拟机（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，MaABBTicore 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. OyeABBTe

OyeABBTe 是一个智能合约分析工具，OyeABBTe 可以用来检测智能合约中常见的 bug，比如 reeABBTrancy、事务排序依赖等等。更方便的是，OyeABBTe 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 可以验证区块链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 是一个用于查找区块链智能合约漏洞的自动化工具，Maian 处理合约的字节码，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 是一个针对区块链虚拟机（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 语言构建区块链合约并调试交易。

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。