CyberCrime Shield
cybercrimeshield.org

Smart Contract Audit Report

BNB Miner V2

Community Airdrop Edition

https://bnbcommunity.net

AUDIT TYPE: **PUBLIC**

YOU CAN CHECK THE VALIDITY USING THE QR CODE OR LINK:

https://cybercrimeshield.org/secure/bnbcommunity

Report ID: 291990

Feb 17, 2022

TABLE OF CONTENTS

SMART CONTRACT

https://bscscan.com/address/0x1ae61982c89c2abf554316dec13515a8999d51fc#code

MIRROR

https://cybercrimeshield.org/secure/uploads/bnbminer-v2.sol

CRC32: 23683631

MD5: E562429371964BEA5E3EF50245E8F5ED

SHA-1: 920F9621CC96727477B3CFAE0FEA0A2166F9C86E

INTRODUCTION

Blockchain platforms, such as Nakamoto's Bitcoin, enable the trade of crypto-currencies between mutually mistrusting parties.

To eliminate the need for trust, Nakomoto designed a peer-to-peer network that enables its peers to agree on the trading transactions.

Smart contracts have shown to be applicable in many domains including financial industry, public sector and cross-industry.

The increased adoption of smart contracts demands strong security guarantees. Unfortunately, it is challenging to create smart contracts that are free of security bugs.

As a consequence, critical vulnerabilities in smart contracts are discovered and exploited every few months.

In turn, these exploits have led to losses reaching billions worth of USD in the past few years.

It is apparent that effective security checks for smart contracts are strictly needed.

Our company provides comprehensive, independent smart contract auditing.

We help stakeholders confirm the quality and security of their smart contracts using our standardized audit process.

The scope of this audit was to analyze and document the BNB Miner V2 contract.

This document is not financial advice, you perform all financial actions on your own responsibility.

AUDIT METHODOLOGY

1. Design Patterns

We inspect the structure of the smart contract, including both manual and automated analysis.

2. Static Analysis

The static analysis is performed using a series of automated tools, purposefully designed to test the security of the contract.

All the issues found by tools were manually checked (rejected or confirmed).

3. Manual Analysis

Contract reviewing to identify common vulnerabilities. Comparing of requirements and implementation. Reviewing of a smart contract for compliance with specified customer requirements. Checking for energy optimization and self-documentation. Running tests of the properties of the smart contract in test net.

ISSUES DISCOVERED

Issues are listed from most critical to least critical. Severity is determined by an assessment of the risk of exploitation or otherwise unsafe behavior.

Severity Levels

Critical - Funds may be allocated incorrectly, lost or otherwise result ina significant loss.
Medium - Affects the ability of the contract to operate.
Low - Minimal impact on operational ability.
Informational - No impact on the contract.

AUDIT SUMMARY

The summary result of the audit performed is presented in the table below

Findings list:

LEVEL	AMOUNT
Critical	0
Medium	0
Low	0
Informational	3

FINDINGS

Informational (3)

1-3 Locked money (not applicable to this version of the contract)

line: 96
(!isContract<missing ';'>
(_owner)&&!
isContract(_taxWallet));

line: 95
_owner,

line: 95
addresspayable_taxWallet

Contracts including at least one function with payable decorator should implement a way to withdraw funds, i.e. call send (recommended), raw_call or selfdestruct function at least once.

CONCLUSION

- Contracts have high code readability
- Gas usage is optimal
- Contracts are fully BSC completable
- No backdoors or overflows are present in the contracts

SOURCE CODE

```
/**
 *Submitted for verification at BscScan.com on 2022-02-11
*/

/**

BSC Defi - BNB Miner V2 - Community Airdrop Edition

  ~ 4% daily interest contract basis.
  ~ 4% Referral Bonus, rewards wil go straight to the wallet.
  ~ 2.5% stacking compound bonus every 12 hours, max of 7 days.
  ~ 0.01 BNB Minimum Investment.
  ~ 36 hours cut off time.
  ~ 12 hours withdraw cooldown.
  ~ 0.1 bnb minimum investment.
  ~ 20 bnb max deposits per wallet.
  ~ 40% feedback tax on 2 consecutive withdrawals.
    *Tax will stay in the contract.


  3.5% tax will be transferred to the community wallet and be aidropped to the
       investors who lost their money from V1.
  1.5% tax will be used for dev fee, which will include system maintenance,
       sharing it with the admins and also for future events.

  NOTE: Changes with in the tax wallets will be considered not until every
       single one investor from V1 is paid out.

*/

// SPDX-License-Identifier: MIT
pragma solidity 0.8.9;

contract Community_V2 {
    using SafeMath for uint256;

    /** base parameters **/
    uint256 public EGGS_TO_HIRE_1MINERS = 2160000; /** 4% **/
    uint256 public EGGS_TO_HIRE_1MINERS_COMPOUND = 1200000; /** 6% **/
    uint256 public REFERRAL = 40;
    uint256 public PERCENTS_DIVIDER = 1000;
    uint256 public TAX = 35;
    uint256 public DEV = 15;
```

```
41.     uint256 public MARKET_EGGS_DIVISOR = 8;
42.     uint256 public MARKET_EGGS_DIVISOR_SELL = 2;
43.
44.     /** investment limit parameters **/
45.     uint256 public MIN_INVEST_LIMIT = 100 * 1e15; /** 0.1 BNB  **/
46.     uint256 public WALLET_DEPOSIT_LIMIT = 20 ether; /** 20 BNB  **/
47.
48.     /** bonus parameters **/
49.     uint256 public COMPOUND_BONUS = 25; /** 2.5% **/
50.     uint256 public COMPOUND_BONUS_MAX_TIMES = 14; /** 14 times / 7 days. **/
51.     uint256 public COMPOUND_STEP = 12 * 60 * 60; /** every 12 hours. **/
52.
53.     /** withdrawal tax parameters **/
54.     uint256 public WITHDRAWAL_TAX = 400;
55.     uint256 public WITHDRAWAL_TAX_DAYS = 2;
56.
57.     /* statistics parameters */
58.     uint256 public totalStaked;
59.     uint256 public totalDeposits;
60.     uint256 public totalCompound;
61.     uint256 public totalRefBonus;
62.     uint256 public totalWithdrawn;
63.
64.     /* miner parameters */
65.     uint256 public marketEggs;
66.     uint256 PSN = 10000;
67.     uint256 PSNH = 5000;
68.     bool public contractStarted;
69.
70.     /** cooldown parameters **/
71.     uint256 public CUTOFF_STEP = 36 * 60 * 60; /** 36 hours  **/
72.     uint256 public WITHDRAW_COOLDOWN = 12 * 60 * 60; /** 12 hours  **/
73.
74.     /* addresses */
75.     address payable public owner;
76.     address payable public taxWallet;
77.
78.     struct User {
79.         uint256 initialDeposit;
80.         uint256 userDeposit;
81.         uint256 miners;
82.         uint256 claimedEggs;
83.         uint256 lastHatch;
84.         address referrer;
85.         uint256 referralsCount;
86.         uint256 referralEggRewards;
```

```
87.           uint256 totalWithdrawn;
88.           uint256 dailyCompoundBonus;
89.           uint256 withdrawCount;
90.           uint256 lastWithdrawTime;
91.       }
92.
93.       mapping(address => User) public users;
94.
95.       constructor(address payable _owner, address payable _taxWallet) {
96.           require(!isContract(_owner) && !isContract(_taxWallet));
97.           owner = _owner;
98.           taxWallet = _taxWallet;
99.       }
100.
101.      function isContract(address addr) internal view returns (bool) {
102.          uint size;
103.          assembly { size := extcodesize(addr) }
104.          return size > 0;
105.      }
106.
107.      function initialize() public{
108.          if (!contractStarted) {
109.              if (msg.sender == owner) {
110.                  require(marketEggs == 0);
111.                  contractStarted = true;
112.                  marketEggs = 216000000000;
113.              } else revert("Contract not yet started.");
114.          }
115.      }
116.
117.      function hatchEggs(bool isCompound) public {
118.          User storage user = users[msg.sender];
119.          require(contractStarted, "Contract not yet Started.");
120.
121.          uint256 eggsUsed = getMyEggs();
122.          uint256 eggsForCompound = eggsUsed;
123.
124.          /**  miner increase -- check if for compound, new deposit and compound
             can have different percentage basis. **/
125.          uint256 newMiners;
126.
127.          /** isCompound -- only true when compounding. **/
128.          if(isCompound) {
129.              uint256 dailyCompoundBonus = getDailyCompoundBonus(msg.sender,
             eggsForCompound);
130.              eggsForCompound = eggsForCompound.add(dailyCompoundBonus);
```

```
131.             uint256 eggsUsedValue = calculateEggSell(eggsForCompound);
132.             user.userDeposit = user.userDeposit.add(eggsUsedValue);
133.             totalCompound = totalCompound.add(eggsUsedValue);
134.             newMiners = eggsForCompound.div(EGGS_TO_HIRE_1MINERS_COMPOUND);
135.         }else{
136.             newMiners = eggsForCompound.div(EGGS_TO_HIRE_1MINERS);
137.         }
138.
139.         /** compounding bonus add count if greater than COMPOUND_STEP. **/
140.         if(block.timestamp.sub(user.lastHatch) >= COMPOUND_STEP) {
141.             if(user.dailyCompoundBonus < COMPOUND_BONUS_MAX_TIMES) {
142.                 user.dailyCompoundBonus = user.dailyCompoundBonus.add(1);
143.             }
144.         }
145.
146.         /** withdraw Count will only reset if last withdraw time is greater than
            or equal to COMPOUND_STEP.
147.             re-use COMPOUND_STEP step time constant to do validation the
            validation  **/
148.         if(block.timestamp.sub(user.lastWithdrawTime) >= COMPOUND_STEP){
149.             user.withdrawCount = 0;
150.         }
151.
152.         user.miners = user.miners.add(newMiners);
153.         user.claimedEggs = 0;
154.         user.lastHatch = block.timestamp;
155.
156.     /** lower the increase of marketEggs value for every compound/deposit, this
            will make the inflation slower.  20%(5) to 8%(12). **/
157.         marketEggs = marketEggs.add(eggsUsed.div(MARKET_EGGS_DIVISOR));
158.     }
159.
160.     function sellEggs() public{
161.         require(contractStarted);
162.         User storage user = users[msg.sender];
163.         uint256 hasEggs = getMyEggs();
164.         uint256 eggValue = calculateEggSell(hasEggs);
165.
166.         if(user.lastHatch.add(WITHDRAW_COOLDOWN) > block.timestamp)
            revert("Withdrawals can only be done after withdraw cooldown.");
167.
168.         /** reset claim. **/
169.         user.claimedEggs = 0;
170.
171.         /** reset hatch time. **/
172.         user.lastHatch = block.timestamp;
```

```
173.
174.          /** reset daily compound bonus. **/
175.          user.dailyCompoundBonus = 0;
176.
177.          /** add withdraw count. **/
178.          user.withdrawCount = user.withdrawCount.add(1);
179.
180.          /** if user withdraw count is >= 2, implement = 40% tax. **/
181.          if(user.withdrawCount >= WITHDRAWAL_TAX_DAYS){
182.            eggValue =
             eggValue.sub(eggValue.mul(WITHDRAWAL_TAX).div(PERCENTS_DIVIDER));
183.          }
184.
185.          /** set last withdrawal time **/
186.          user.lastWithdrawTime = block.timestamp;
187.
188.          /** lowering the amount of eggs that is being added to the total eggs
             supply to only 5% for each sell **/
189.          marketEggs = marketEggs.add(hasEggs.div(MARKET_EGGS_DIVISOR_SELL));
190.
191.          /** check if contract has enough funds to pay -- one last ride. **/
192.          if(getBalance() < eggValue) {
193.              eggValue = getBalance();
194.          }
195.
196.          uint256 eggsPayout = eggValue.sub(payFees(eggValue));
197.          payable(address(msg.sender)).transfer(eggsPayout);
198.          user.totalWithdrawn = user.totalWithdrawn.add(eggsPayout);
199.          totalWithdrawn = totalWithdrawn.add(eggsPayout);
200.      }
201.
202.      /** transfer amount of BNB **/
203.      function buyEggs(address ref) public payable{
204.          User storage user = users[msg.sender];
205.          require(msg.value >= MIN_INVEST_LIMIT, "Mininum investment not met.");
206.          require(user.initialDeposit.add(msg.value) <= WALLET_DEPOSIT_LIMIT, "Max
             deposit limit reached.");
207.          uint256 eggsBought = calculateEggBuy(msg.value,
             address(this).balance.sub(msg.value));
208.          user.userDeposit = user.userDeposit.add(msg.value);
209.          user.initialDeposit = user.initialDeposit.add(msg.value);
210.          user.claimedEggs = user.claimedEggs.add(eggsBought);
211.
212.          if (user.referrer == address(0)) {
213.              if (ref != msg.sender) {
214.                  user.referrer = ref;
```

```
215.             }
216.
217.             address upline1 = user.referrer;
218.             if (upline1 != address(0)) {
219.                 users[upline1].referralsCount =
        users[upline1].referralsCount.add(1);
220.             }
221.         }
222.
223.         if (user.referrer != address(0)) {
224.             address upline = user.referrer;
225.             if (upline != address(0)) {
226.                 /** referral rewards will be in BNB **/
227.                 uint256 refRewards =
        msg.value.mul(REFERRAL).div(PERCENTS_DIVIDER);
228.                 payable(address(upline)).transfer(refRewards);
229.                 /** referral rewards will be in BNB value **/
230.                 users[upline].referralEggRewards =
        users[upline].referralEggRewards.add(refRewards);
231.                 totalRefBonus = totalRefBonus.add(refRewards);
232.             }
233.         }
234.
235.         uint256 eggsPayout = payFees(msg.value);
236.         /** less the fee on total Staked to give more transparency of data. **/
237.         totalStaked = totalStaked.add(msg.value.sub(eggsPayout));
238.         totalDeposits = totalDeposits.add(1);
239.         hatchEggs(false);
240.     }
241.
242.     function payFees(uint256 eggValue) internal returns(uint256){
243.         uint256 tax = eggValue.mul(TAX).div(PERCENTS_DIVIDER);
244.         uint256 dev = eggValue.mul(DEV).div(PERCENTS_DIVIDER);
245.         taxWallet.transfer(tax);
246.         owner.transfer(dev);
247.         return tax.add(dev);
248.     }
249.
250.     function getDailyCompoundBonus(address _adr, uint256 amount) public view
        returns(uint256){
251.         if(users[_adr].dailyCompoundBonus == 0) {
252.             return 0;
253.         } else {
254.             uint256 totalBonus =
        users[_adr].dailyCompoundBonus.mul(COMPOUND_BONUS);
255.             uint256 result = amount.mul(totalBonus).div(PERCENTS_DIVIDER);
```

```
256.             return result;
257.         }
258.     }
259.
260.     function getUserInfo(address _adr) public view returns(uint256
            _initialDeposit, uint256 _userDeposit, uint256 _miners,
261.      uint256 _claimedEggs, uint256 _lastHatch, address _referrer, uint256
            _referrals,
262.      uint256 _totalWithdrawn, uint256 _referralEggRewards, uint256
            _dailyCompoundBonus, uint256 _lastWithdrawTime, uint256 _withdrawCount) {
263.         _initialDeposit = users[_adr].initialDeposit;
264.         _userDeposit = users[_adr].userDeposit;
265.         _miners = users[_adr].miners;
266.         _claimedEggs = users[_adr].claimedEggs;
267.         _lastHatch = users[_adr].lastHatch;
268.         _referrer = users[_adr].referrer;
269.         _referrals = users[_adr].referralsCount;
270.         _totalWithdrawn = users[_adr].totalWithdrawn;
271.         _referralEggRewards = users[_adr].referralEggRewards;
272.         _dailyCompoundBonus = users[_adr].dailyCompoundBonus;
273.         _lastWithdrawTime = users[_adr].lastWithdrawTime;
274.         _withdrawCount = users[_adr].withdrawCount;
275.     }
276.
277.     function getBalance() public view returns(uint256){
278.         return address(this).balance;
279.     }
280.
281.     function getTimeStamp() public view returns (uint256) {
282.         return block.timestamp;
283.     }
284.
285.     function getAvailableEarnings(address _adr) public view returns(uint256) {
286.         uint256 userEggs =
            users[_adr].claimedEggs.add(getEggsSinceLastHatch(_adr));
287.         return calculateEggSell(userEggs);
288.     }
289.
290.     function calculateTrade(uint256 rt,uint256 rs, uint256 bs) public view
            returns(uint256){
291.         return SafeMath.div(SafeMath.mul(PSN, bs), SafeMath.add(PSNH,
            SafeMath.div(SafeMath.add(SafeMath.mul(PSN, rs), SafeMath.mul(PSNH, rt)),
            rt)));
292.     }
293.
294.     function calculateEggSell(uint256 eggs) public view returns(uint256){
```

```
295.          return calculateTrade(eggs, marketEggs, getBalance());
296.      }
297.
298.      function calculateEggBuy(uint256 eth,uint256 contractBalance) public view
              returns(uint256){
299.          return calculateTrade(eth, contractBalance, marketEggs);
300.      }
301.
302.      function calculateEggBuySimple(uint256 eth) public view returns(uint256){
303.          return calculateEggBuy(eth, getBalance());
304.      }
305.
306.      /** How many miners and eggs per day user will recieve based on BNB deposit
              **/
307.      function getEggsYield(uint256 amount) public view returns(uint256,uint256) {
308.          uint256 eggsAmount = calculateEggBuy(amount ,
              getBalance().add(amount).sub(amount));
309.          uint256 miners = eggsAmount.div(EGGS_TO_HIRE_1MINERS);
310.          uint256 day = 1 days;
311.          uint256 eggsPerDay = day.mul(miners);
312.          uint256 earningsPerDay = calculateEggSellForYield(eggsPerDay, amount);
313.          return(miners, earningsPerDay);
314.      }
315.
316.      function calculateEggSellForYield(uint256 eggs,uint256 amount) public view
              returns(uint256){
317.          return calculateTrade(eggs,marketEggs, getBalance().add(amount));
318.      }
319.
320.      function getSiteInfo() public view returns (uint256 _totalStaked, uint256
              _totalDeposits, uint256 _totalCompound, uint256 _totalRefBonus) {
321.          return (totalStaked, totalDeposits, totalCompound, totalRefBonus);
322.      }
323.
324.      function getMyMiners() public view returns(uint256){
325.          return users[msg.sender].miners;
326.      }
327.
328.      function getMyEggs() public view returns(uint256){
329.          return
              users[msg.sender].claimedEggs.add(getEggsSinceLastHatch(msg.sender));
330.      }
331.
332.      function getEggsSinceLastHatch(address adr) public view returns(uint256){
333.          uint256 secondsSinceLastHatch =
              block.timestamp.sub(users[adr].lastHatch);
```

```
334.                                /** get min time. **/
335.         uint256 cutoffTime = min(secondsSinceLastHatch, CUTOFF_STEP);
336.         uint256 secondsPassed = min(EGGS_TO_HIRE_1MINERS, cutoffTime);
337.         return secondsPassed.mul(users[adr].miners);
338.     }
339.
340.     function min(uint256 a, uint256 b) private pure returns (uint256) {
341.         return a < b ? a : b;
342.     }
343.
344.     /** wallet addresses setters **/
345.     function CHANGE_OWNERSHIP(address value) external {
346.         require(msg.sender == owner, "Admin use only.");
347.         owner = payable(value);
348.     }
349.
350.     function CHANGE_FEE_WALLET(address value) external {
351.         require(msg.sender == owner, "Admin use only.");
352.         taxWallet = payable(value);
353.     }
354.
355.     /** percentage setters **/
356.
357.     // 2592000 - 3%, 2160000 - 4%, 1728000 - 5%, 1440000 - 6%, 1200000 - 7%,
        1080000 - 8%
358.     // 959000 - 9%, 864000 - 10%, 720000 - 12%, 575424 - 15%, 540000 - 16%,
        479520 - 18%
359.
360.     function PRC_EGGS_TO_HIRE_1MINERS(uint256 value) external {
361.         require(msg.sender == owner, "Admin use only.");
362.         require(value >= 479520 && value <= 2592000); /** min 3% max 12%**/
363.         EGGS_TO_HIRE_1MINERS = value;
364.     }
365.
366.     function PRC_EGGS_TO_HIRE_1MINERS_COMPOUND(uint256 value) external {
367.         require(msg.sender == owner, "Admin use only.");
368.         require(value >= 479520 && value <= 2592000); /** min 3% max 12%**/
369.         EGGS_TO_HIRE_1MINERS_COMPOUND = value;
370.     }
371.
372.     function PRC_TAX(uint256 value) external {
373.         require(msg.sender == owner, "Admin use only.");
374.         require(value >= 10 && value <= 50); /** 5% max **/
375.         TAX = value;
376.     }
377.
```

```
378.     function PRC_DEV(uint256 value) external {
379.         require(msg.sender == owner, "Admin use only.");
380.         require(value <= 50); /** 5% max **/
381.         DEV = value;
382.     }
383.
384.     function PRC_REFERRAL(uint256 value) external {
385.         require(msg.sender == owner, "Admin use only.");
386.         require(value >= 10 && value <= 70); /** 70% max **/
387.         REFERRAL = value;
388.     }
389.
390.     function PRC_MARKET_EGGS_DIVISOR(uint256 value) external {
391.         require(msg.sender == owner, "Admin use only.");
392.         require(value <= 50); /** 50 = 2% as lowest **/
393.         MARKET_EGGS_DIVISOR = value;
394.     }
395.
396.     function PRC_MARKET_EGGS_DIVISOR_SELL(uint256 value) external {
397.         require(msg.sender == owner, "Admin use only.");
398.         require(value <= 50); /** 50 = 2% as lowest **/
399.         MARKET_EGGS_DIVISOR_SELL = value;
400.     }
401.
402.     /** withdrawal tax setters **/
403.     function SET_WITHDRAWAL_TAX(uint256 value) external {
404.         require(msg.sender == owner, "Admin use only.");
405.         require(value <= 500); /** Max Tax is 50% or lower **/
406.         WITHDRAWAL_TAX = value;
407.     }
408.
409.     function SET_WITHDRAW_DAYS_TAX(uint256 value) external {
410.         require(msg.sender == owner, "Admin use only.");
411.         require(value >= 2); /** Minimum 3 days **/
412.         WITHDRAWAL_TAX_DAYS = value;
413.     }
414.
415.     /** bonus setters **/
416.     function BONUS_DAILY_COMPOUND(uint256 value) external {
417.         require(msg.sender == owner, "Admin use only.");
418.         require(value >= 10 && value <= 900); /** 90% max **/
419.         COMPOUND_BONUS = value;
420.     }
421.
422.     function BONUS_DAILY_COMPOUND_BONUS_MAX_TIMES(uint256 value) external {
423.         require(msg.sender == owner, "Admin use only.");
```

```
424.          require(value <= 30); /** 30 max **/
425.          COMPOUND_BONUS_MAX_TIMES = value;
426.      }
427.
428.      function BONUS_COMPOUND_STEP(uint256 value) external {
429.          require(msg.sender == owner, "Admin use only.");
430.           /** hour conversion **/
431.          COMPOUND_STEP = value * 60 * 60;
432.      }
433.
434.      function SET_INVEST_MIN(uint256 value) external {
435.          require(msg.sender == owner, "Admin use only");
436.          MIN_INVEST_LIMIT = value * 1e15;
437.      }
438.
439.      /** time setters **/
440.      function SET_CUTOFF_STEP(uint256 value) external {
441.          require(msg.sender == owner, "Admin use only");
442.          CUTOFF_STEP = value * 60 * 60;
443.      }
444.
445.      function SET_WITHDRAW_COOLDOWN(uint256 value) external {
446.          require(msg.sender == owner, "Admin use only");
447.          require(value <= 24);
448.          WITHDRAW_COOLDOWN = value * 60 * 60;
449.      }
450.
451.      function SET_WALLET_DEPOSIT_LIMIT(uint256 value) external {
452.          require(msg.sender == owner, "Admin use only");
453.          require(value >= 10);
454.          WALLET_DEPOSIT_LIMIT = value * 1 ether;
455.      }
456. }
457.
458. library SafeMath {
459.
460.    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
461.      if (a == 0) {
462.        return 0;
463.      }
464.      uint256 c = a * b;
465.      assert(c / a == b);
466.      return c;
467.    }
468.
469.    function div(uint256 a, uint256 b) internal pure returns (uint256) {
```

```
470.     uint256 c = a / b;
471.     return c;
472.   }
473.
474.   function sub(uint256 a, uint256 b) internal pure returns (uint256) {
475.     assert(b <= a);
476.     return a - b;
477.   }
478.
479.   function add(uint256 a, uint256 b) internal pure returns (uint256) {
480.     uint256 c = a + b;
481.     assert(c >= a);
482.     return c;
483.   }
484.
485.   function mod(uint256 a, uint256 b) internal pure returns (uint256) {
486.     require(b != 0);
487.     return a % b;
488.   }
489. }
```