以太坊智能合约审计CheckList

作者：知道创宇404区块链安全研究团队

时间：2018.11.12

在以太坊合约审计checkList中，我将以太坊合约审计中遇到的问题分为5大种，包括编码规范问题、设计缺陷问题、编码安全问题、编码设计问题、编码问题隐患。其中涵盖了超过29种会出现以太坊智能合约审计过程中遇到的问题。帮助智能合约的开发者和安全工作者快速入门智能合约安全。

本CheckList在完成过程中参考并整理兼容了各大区块链安全研究团队的研究成果，CheckList中如有不完善/错误的地方也欢迎大家提issue.

由于本文的目的主要是**CheckList**，所以文中不会包含太详细的漏洞/隐患信息，大部分漏洞分析在扫描报告中会有所提及。

以太坊智能合约审计CheckList 目录

- 以太坊智能合约审计CheckList
- 以太坊合约审计checkList
- 1、编码规范问题
  - (1) 编译器版本
  - (2) 构造函数书写问题
  - (3) 返回标准
  - (4) 事件标准
  - (5) 假充值问题
- 2、设计缺陷问题
  - (1) approve授权函数条件竞争
  - (2) 循环Dos问题
    - [1] 循环消耗问题
      - 真实世界事件
    - [2] 循环安全问题
- 3、编码安全问题
  - (1) 溢出问题
    - [1] 算术溢出
      - 真实世界事件
    - [2] 铸币烧币溢出问题
      - 真实世界事件
  - (2) 重入漏洞
    - 真实世界事件
  - (3) call注入
    - 真实世界事件
  - (4) 权限控制
    - 真实世界事件
  - (5) 重放攻击
- 4、编码设计问题
  - (1) 地址初始化问题
  - (2) 判断函数问题
  - (3) 余额判断问题

    - (4) 转账函数问题
    - (5) 代码外部调用设计问题
    - (6) 错误处理
    - (7) 弱随机数问题
        - 真实世界事件
    - (8) 变量覆盖问题
- 5、编码问题隐患
    - (1) 语法特性问题
    - (2) 数据私密问题
    - (3) 数据可靠性
    - (4) gas消耗优化
    - (5) 合约用户
    - (6) 日志记录
    - (7) 回调函数
    - (8) Owner权限问题
    - (9) 用户鉴权问题
    - (10) 条件竞争问题
        - 真实世界事件
    - (11) 未初始化的储存指针
    - (12) 「输入假名」问题
        - 真实世界事件
- 以太坊合约审计checkList审计系列报告
- REF

1、编码规范问题

(1) 编译器版本

合约代码中，应指定编译器版本。建议使用最新的编译器版本

```
pragma solidity ^0.4.25;
```

老版本的编译器可能会导致各种已知的安全问题，例如https://paper.seebug.org/631/#44-dividenddistributor

v0.4.23更新了一个编译器漏洞，在这个版本中如果同时使用了两种构造函数，即

```
contract a {
    function a() public{
        ...
    }
    constructor() public{
        ...
    }
}
```

会忽略其中的一个构造函数，该问题只影响v0.4.22

v0.4.25修复了下面提到的未初始化存储指针问题。

https://etherscan.io/solcbuginfo

(2) 构造函数书写问题

对应不同编译器版本应使用正确的构造函数，否则可能导致合约所有者变更

在小于0.4.22版本的solidify编译器语法要求中，合约构造函数必须和合约名字相等， 名字受到大小写影响。如：

```
contract Owned {
    function Owned() public{
    }
```

在0.4.22版本以后，引入了constructor关键字作为构造函数声明，但不需要function

```
contract Owned {
    constructor() public {
    }
```

如果没有按照对应的写法，构造函数就会被编译成一个普通函数，可以被任意人调用，会导致owner权限被窃取等更严重的后果。

(3) 返回标准

遵循ERC20规范，要求transfer、approve函数应返回bool值，需要添加返回值代码

```
function transfer(address _to, uint256 _value) public returns (bool success)
```

而transferFrom返回结果应该和transfer返回结果一致。

(4) 事件标准

遵循ERC20规范，要求transfer、approve函数触发相应的事件

```
function approve(address _spender, uint256 _value) public returns (bool success){
    allowance[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value)
    return true
```

(5) 假充值问题

转账函数中，对余额以及转账金额的判断，需要使用require函数抛出错误，否则会错误的判断为交易成功

```
function transfer(address _to, uint256 _value) returns (bool success) {
    if (balances[msg.sender] >= _value && _value > 0) {
        balances[msg.sender] -= _value;
        balances[_to] += _value;
```

```
        Transfer(msg.sender, _to, _value);
        return true;
    } else { return false; }
}
```

上述代码可能会导致假充值。

正确代码如下：

```
function transfer(address _to, uint256 _amount)  public returns (bool success) {
    require(_to != address(0));
    require(_amount <= balances[msg.sender]);

    balances[msg.sender] = balances[msg.sender].sub(_amount);
    balances[_to] = balances[_to].add(_amount);
    emit Transfer(msg.sender, _to, _amount);
    return true;
}
```

2、设计缺陷问题

(1) approve授权函数条件竞争

approve函数中应避免条件竞争。在修改allowance前，应先修改为0，再修改为_value。

这个漏洞的起因是由于底层矿工协议中为了鼓励矿工挖矿，矿工可以自己决定打包什么交易，为了收益更大，矿工一般会选择打包gas price更大的交易，而不会依赖交易顺序的前后。

通过置0的方式，可以在一定程度上缓解条件竞争中产生的危害，合约管理人可以通过检查日志来判断是否有条件竞争情况的发生，这种修复方式更大的意义在于，提醒使用approve函数的用户，该函数的操作在一定程度上是不可逆的。

```
function approve(address _spender, uint256 _value) public returns (bool success){
    allowance[msg.sender][_spender] = _value;
    return true
```

上述代码就有可能导致条件竞争。

应在approve中加入

```
require((_value == 0) || (allowance[msg.sender][_spender] == 0));
```

将allowance先改为0再改为对应数字

(2) 循环Dos问题

[1] 循环消耗问题

在合约中，不推荐使用太大次的循环

在以太坊中，每一笔交易都会消耗一定量的gas，而实际消耗量是由交易的复杂度决定的，循环次数越大，交易的复杂度越高，当超过允许的最大gas消耗量时，会导致交易失败。

真实世界事件

Simoleon (SIM)

- https://paper.seebug.org/646/

Pandemica

- https://bcsec.org/index/detail/id/260/tag/2

[2] 循环安全问题

合约中，应尽量避免循环次数受到用户控制，攻击者可能会使用过大的循环来完成Dos攻击

当用户需要同时向多个账户转账，我们需要对目标账户列表遍历转账，就有可能导致Dos攻击。

```
function Distribute(address[] _addresses, uint256[] _values) payable returns(bool){
    for (uint i = 0; i < _addresses.length; i++) {
        transfer(_addresses[i], _values[i]);
    }
    return true;
}
```

遇到上述情况是，推荐使用withdrawFunds来让用户取回自己的代币，而不是发送给对应账户，可以在一定程序上减少危害。

上述代码如果控制函数调用，那么就可以构造巨大循环消耗gas，造成Dos问题

3、编码安全问题

(1) 溢出问题

[1] 算术溢出

在调用加减乘除时，应使用safeMath库来替代，否则容易导致算数上下溢，造成不可避免的损失

```
pragma solidity ^0.4.18;

contract Token {

  mapping(address => uint) balances;
  uint public totalSupply;

  function Token(uint _initialSupply) {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint _value) public returns (bool) {
    require(balances[msg.sender] - _value >= 0); //可以通过下溢来绕过判断
    balances[msg.sender] -= _value;
    balances[_to] += _value;
```

```
    return true;
  }

  function balanceOf(address _owner) public constant returns (uint balance) {
    return balances[_owner];
  }
}
```

`balances[msg.sender] - _value >= 0`可以通过下溢来绕过判断。

通常的修复方式都是使用openzeppelin-safeMath，但也可以通过对不同变量的判断来限制，但很难对乘法和指数做什么限制。

正确的写法如下：

```
function transfer(address _to, uint256 _amount)  public returns (bool success) {
    require(_to != address(0));
    require(_amount <= balances[msg.sender]);

    balances[msg.sender] = balances[msg.sender].sub(_amount);
    balances[_to] = balances[_to].add(_amount);
    emit Transfer(msg.sender, _to, _amount);
    return true;
}
```

真实世界事件

Hexagon

- 代币变泡沫，以太坊Hexagon溢出漏洞比狗庄还过分

SMT/BEC

- Solidity合约中的整数安全问题——SMT/BEC合约整数溢出解析

[2] 铸币烧币溢出问题

铸币函数中，应对totalSupply设置上限，避免因为算术溢出等漏洞导致恶意铸币增发

```
function TokenERC20(
    uint256 initialSupply,
    string tokenName,
    string tokenSymbol
) public {
    totalSupply = initialSupply * 10 ** uint256(decimals);
    balanceOf[msg.sender] = totalSupply;
    name = tokenName;
    symbol = tokenSymbol;
}
```

上述代码中就未对totalSupply做限制，可能导致指数算数上溢。

正确写法如下：

```
contract OPL {
    // Public variables
    string public name;
    string public symbol;
    uint8 public decimals = 18; // 18 decimals
    bool public adminVer = false;
    address public owner;
    uint256 public totalSupply;
    function OPL() public {
        totalSupply = 210000000 * 10 ** uint256(decimals);
        ...
}
```

真实世界事件

- ERC20 智能合约整数溢出系列漏洞披露

(2) 重入漏洞

智能合约中避免使用call来交易，避免重入漏洞

在智能合约中提供了call、send、transfer三种方式来交易以太坊，其中call最大的区别就是没有限制gas，而其他两种在gas不够的情况下都会报out of gas。

重入漏洞有几大特征。
1、使用了call函数作为转账函数
2、没有限制call函数的gas
3、扣余额在转账之后
4、call时加入了()来执行fallback函数

```
function withdraw(uint _amount) {
        require(balances[msg.sender] >= _amount);
        msg.sender.call.value(_amount)();
        balances[msg.sender] -= _amount;
}
```

上述代码就是一个简单的重入漏洞的demo。通过重入注入转账，将大量合约代币递归转账而出。

对于可能存在的重入问题，尽可能的使用transfer函数完成转账，或者限制call执行的gas，都可以有效的减少该问题的危害。

```
contract EtherStore {

    // initialise the mutex
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
```

```
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // set the reEntrancy mutex before the external call
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // release the mutex after the external call
        reEntrancyMutex = false;
    }
}
```

上述代码是一种用互斥锁来避免递归防护方式。

真实世界事件

The Dao

- The DAO
- The DAO address

(3) call注入

call函数调用时，应该做严格的权限控制，或直接写死call调用的函数

在EVM的设计中，如果call的参数data是0xdeadbeef(假设的一个函数名) + 0x0000000000.....01，这样的话就是调用函数

call注入可能导致代币窃取，权限绕过，通过call注入可以调用私有函数，甚至部分高权限函数。

```
addr.call(data);
addr.delegatecall(data);
addr.callcode(data);
```

如delegatecall，在合约内必须调用其它合约时，可以使用关键字library，这样可以确保合约是无状态而且不可自毁的。通过强制设置合约为无状态可以一定程度上缓解储存环境的复杂性，防止攻击者通过修改状态来攻击合约。

真实世界事件

call注入

- 以太坊智能合约call注入攻击
- 以太坊 Solidity 合约 call 函数簇滥用导致的安全风险

(4) 权限控制

合约中不同函数应设置合理的权限

检查合约中各函数是否正确使用了public、 private等关键词进行可见性修饰，检查合约是否正确定义并使用了modifier对关键函数进行访问限制，避免越权导致的问题。

```
function initContract() public {
    owner = msg.sender;
}
```

上述代码作为初始函数不应该为public。

真实世界事件

Parity Multi-sig bug 1

- Parity Multi-sig bug 1

Parity Multi-sig bug 2

- Parity Multi-sig bug 2

Rubixi

- Rubixi

(5) 重放攻击

合约中如果涉及委托管理的需求，应注意验证的不可复用性，避免重放攻击

在资产管理体系中，常有委托管理的情况，委托人将资产给受托人管理，委托人支付一定的费用给受托人。这个业务场景在智能合约中也比较普遍。

这里举例子为transferProxy函数，该函数用于当user1转token给user3，但没有eth来支付gasprice，所以委托user2代理支付，通过调用transferProxy来完成。

```
function transferProxy(address _from, address _to, uint256 _value, uint256 _fee,
    uint8 _v, bytes32 _r, bytes32 _s) public returns (bool){

    if(balances[_from] < _fee + _value
        || _fee > _fee + _value) revert();

    uint256 nonce = nonces[_from];
    bytes32 h = keccak256(_from,_to,_value,_fee,nonce,address(this));
    if(_from != ecrecover(h,_v,_r,_s)) revert();

    if(balances[_to] + _value < balances[_to]
        || balances[msg.sender] + _fee < balances[msg.sender]) revert();
    balances[_to] += _value;
    emit Transfer(_from, _to, _value);

    balances[msg.sender] += _fee;
    emit Transfer(_from, msg.sender, _fee);

    balances[_from] -= _value + _fee;
    nonces[_from] = nonce + 1;
    return true;
}
```

这个函数的问题在于nonce值是可以预判的，其他变量不变的情况下，可以进行重放攻击，多次转账。

漏洞来自于Defcon2018演讲议题

Replay Attacks on Ethereum Smart Contracts
Replay Attacks on Ethereum Smart Contracts pdf

4、编码设计问题

(1) 地址初始化问题

涉及到地址的函数中，建议加入`require(_to!=address(0))`验证，有效避免用户误操作或未知错误导致的不必要的损失

由于EVM在编译合约代码时初始化的地址为0，如果开发者在代码中初始化了某个address变量，但未赋予初值，或用户在发起某种操作时，误操作未赋予address变量值，但在下面的代码中操作了这个变量，就可能导致不必要的安全风险。

这样的检查可以以最简单的方式避免未知错误、短地址攻击等问题的发生。

(2) 判断函数问题

及到条件判断的地方，使用require函数而不是assert函数，因为assert会导致剩余的gas全部消耗掉，而他们在其他方面的表现都是一致的

值得注意的是，assert存在强制一致性，对于固定变量的检查来说，assert可以用于避免一些未知的问题，因为他会强制终止合约并使其无效化，在一些固定条件下，assert更适用。

(3) 余额判断问题

不要假设合约创建时余额为0，可以强制转账

谨慎编写用于检查账户余额的不变量，因为攻击者可以强制发送wei到任何账户，即使fallback函数throw也不行。

攻击者可以用1wei来创建合约，然后调用`selfdestruct(victimAddress)`来销毁，这样余额就会强制转移给目标，而且目标合约没有代码执行，无法阻止。

值得注意的是，在打包过程中，攻击者可以通过条件竞争在合约创建前转账，这样在合约创建时余额就不为0.

(4) 转账函数问题

在完成交易时，默认情况下推荐使用transfer而不是send完成交易

当transfer或者send函数的目标是合约时，会调用合约的fallback函数，但fallback函数执行失败时。

transfer会抛出错误并自动回滚，而send会返回false，所以在使用send时需要判断返回类型，否则可能会导致转账失败但余额减少的情况。

```
function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
}
```

上面给出的代码中使用 send() 函数进行转账，因为这里没有验证 send() 返回值，如果msg.sender 为合约账户 fallback() 调用失败，则 send() 返回false，最终导致账户余额减少了，钱却没有拿到。

(5) 代码外部调用设计问题

对于外部合约优先使用pull而不是push

在进行外部调用时，总会有意无意的失败，为了避免发生未知的损失，应该经可能的把对外的操作改为用户自己来取。
错误样例:

```
contract auction {
    address highestBidder;
    uint highestBid;

    function bid() payable {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            if (!highestBidder.send(highestBid)) { // 可能会发生错误
                throw;
            }
        }

       highestBidder = msg.sender;
       highestBid = msg.value;
    }
}
```

当需要向某一方转账时，将转账改为定义withdraw函数，让用户自己来执行合约将余额取出，这样可以最大程度的避免未知的损失。

范例代码:

```
contract auction {
    address highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() payable external {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            refunds[highestBidder] += highestBid; // 记录在refunds中
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        if (!msg.sender.send(refund)) {
            refunds[msg.sender] = refund; // 如果转账错误还可以挽回
        }
    }
}
```

(6) 错误处理

合约中涉及到call等在address底层操作的方法时，做好合理的错误处理

```
address.call()
address.callcode()
address.delegatecall()
address.send()
```

这类操作如果遇到错误并不会抛出异常，而是会返回false并继续执行。

```
function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
}
```

上述代码没有校验send的返回值，如果msg.sender是合约账户，fallback调用失败时，send返回false。

所以当使用上述方法时，需要对返回值做检查并做错误处理。

```
if(!someAddress.send(55)) {
    // Some failure code
}
```

https://paper.seebug.org/607/#4-unchecked-return-values-for-low-level-calls

值得注意的一点是，作为EVM设计的一部分，下面这些函数如果调用的合约不存在，将会返回True

```
call、delegatecall、callcode、staticcall
```

在调用这类函数之前，需要对地址的有效性做检查。

(7) 弱随机数问题

智能合约上随机数生成方式需要更多考量

Fomo3D合约在空投奖励的随机数生成中就引入了block信息作为随机数种子生成的参数，导致随机数种子只受到合约地址影响，无法做到完全随机。

```
function airdrop()
    private
    view
    returns(bool)
{
    uint256 seed = uint256(keccak256(abi.encodePacked(
```

```
        (block.timestamp).add
        (block.difficulty).add
        ((uint256(keccak256(abi.encodePacked(block.coinbase)))) / (now)).add
        (block.gaslimit).add
        ((uint256(keccak256(abi.encodePacked(msg.sender)))) / (now)).add
        (block.number)

    )));
    if((seed - ((seed / 1000) * 1000)) < airDropTracker_)
        return(true);
    else
        return(false);
}
```

上述这段代码直接导致了Fomo3d薅羊毛事件的诞生。真实世界损失巨大，超过数千eth。

所以在合约中关于这样的应用时，考虑更合适的生成方式和合理的利用顺序非常重要。

这里提供一个比较合理的随机数生成方式**hash-commit-reveal**，即玩家提交行动计划，然后行动计划hash后提交给后端，后端生成相应的hash值，然后生成对应的随机数reveal，返回对应随机数commit。这样，服务端拿不到行动计划，客户端也拿不到随机数。

有一个很棒的实现代码是dice2win的随机数生成代码。

但**hash-commit-reveal**最大的问题在于服务端会在用户提交之后短暂的获得整个过程中的所有数据，如果恶意进行选择中止攻击，也在一定程度上破坏了公平性。详细分析见智能合约游戏之殇——Dice2win安全分析

当然**hash-commit**在一些简单场景下也是不错的实现方式。即玩家提交行动计划的hash，然后生成随机数，然后提交行动计划。

真实世界事件

Fomo3d薅羊毛

- https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide/
- 8万笔交易「封死」以太坊网络，只为抢夺Fomo3D大奖?

Last Winner

- https://paper.seebug.org/672/

(8) 变量覆盖问题

在合约中避免array变量key可以被控制

```
map[uint256(msg.sender)+x] = blockNum;
```

在EVM中数组和其他类型不同，因为数组时动态大小的，所以数组类型的数据计算方式为

```
address(map_data) = sha3(key)+offset
```

其中key就是map变量定义的位置，也就是1，offset就是数组中的偏移，比如map[2]，offset就是2.

map[2]的地址就是`sha3(1)+2`，假设map[2]=2333，则`storage[sha3(1)+2]=2333`。

这样一来就出现问题了，由于offset我们可控，我们就可以向storage的任意地址写值。

这就可能覆盖storage的任意地址的值，影响代码本身的逻辑，导致进一步更严重的问题。

详细的原理可以看

- 以太坊智能合约 OPCODE 逆向之理论基础篇
- https://paper.seebug.org/739/

5、编码问题隐患

(1) 语法特性问题

在智能合约中小心整数除法的向下取整问题

在智能合约中，所有的整数除法都会向下取整到最接近的整数，当我们需要更高的精度时，我们需要使用乘数来加大这个数字。

该问题如果在代码中显式出现，编译器会提出问题警告，无法继续编译，但如果隐式出现，将会采取向下取整的处理方式。

错误样例

```
uint x = 5 / 2; // 2
```

正确代码

```
uint multiplier = 10;
uint x = (5 * multiplier) / 2;
```

(2) 数据私密问题

注意链上的所有数据都是公开的

在合约中，所有的数据包括私有变量都是公开的，不可以将任何有私密性的数据储存在链上。

(3) 数据可靠性

合约中不应该让时间戳参与到代码中，容易受到矿工的干扰，应使用block.height等不变的数据

uint someVariable = now + 1;

if (now % 2 == 0) { // now可能被矿工控制

}

(4) gas消耗优化

对于某些不涉及状态变化的函数和变量可以加constant来避免gas的消耗

```
contract EUXLinkToken is ERC20 {
    using SafeMath for uint256;
    address owner = msg.sender;

    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;
    mapping (address => bool) public blacklist;
```

```
    string public constant name = "xx";
    string public constant symbol = "xxx";
    uint public constant decimals = 8;
    uint256 public totalSupply = 1000000000e8;
    uint256 public totalDistributed = 200000000e8;
        uint256 public totalPurchase = 200000000e8;
    uint256 public totalRemaining = totalSupply.sub(totalDistributed).sub(totalPurchase);

    uint256 public value = 5000e8;
        uint256 public purchaseCardinal = 5000000e8;

        uint256 public minPurchase = 0.001e18;
        uint256 public maxPurchase = 10e18;
```

(5) 合约用户

合约中，应尽量考虑交易目标为合约时的情况，避免因此产生的各种恶意利用

```
contract Auction{
    address public currentLeader;
    uint256 public hidghestBid;

    function bid() public payable {
        require(msg.value > highestBid);
        require(currentLeader.send(highestBid));
        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

上述合约就是一个典型的没有考虑合约为用户时的情况，这是一个简单的竞拍争夺王位的代码。当交易ether大于合约内的highestBid，当前用户就会成为合约当前的"王"，他的交易额也会成为新的highestBid。

```
contract Attack {
    function () { revert(); }

    function Attack(address _target) payable {
        _target.call.value(msg.value)(bytes4(keccak256("bid()")));
    }
}
```

但当新的用户试图成为新的“王”时，当代码执行到`require(currentLeader.send(highestBid));`时，合约中的fallback函数会触发，如果攻击者在fallback函数中加入`revert()`函数，那么交易就会返回false，即永远无法完成交易，那么当前合约就会一直成为合约当前的"王"。

(6) 日志记录

关键事件应有Event记录，为了便于运维监控，除了转账，授权等函数以外，其他操作也需要加入详细的事件记录，如转移管理员权限、其他特殊的主功能

```
function transferOwnership(address newOwner) onlyOwner public {
    owner = newOwner;
    emit OwnershipTransferred(owner, newowner);
    }
```

(7) 回调函数

合约中定义Fallback函数，并使Fallback函数尽可能的简单

Fallback会在合约执行发生问题时调用（如没有匹配的函数时），而且当调用 `send` 或者 `transfer` 函数时，只有2300gas 用于失败后fallback函数执行,2300 gas只允许执行一组字节码指令，需要谨慎编写，以免gas不够用。

部分样例:

```
function() payable { LogDepositReceived(msg.sender); }

function() public payable{ revert();};
```

(8) Owner权限问题

避免owner权限过大

部分合约owner权限过大，owner可以随意操作合约内各种数据，包括修改规则，任意转账，任意铸币烧币，一旦发生安全问题，可能会导致严重的结果。

关于owner权限问题，应该遵循几个要求:
1、合约创造后，任何人不能改变合约规则，包括规则参数大小等
2、只允许owner从合约中提取余额

(9) 用户鉴权问题

合约中不要使用tx.origin做鉴权

tx.origin代表最初始的地址，如果用户a通过合约b调用了合约c，对于合约c来说，tx.origin就是用户a，而msg.sender才是合约b，对于鉴权来说，这是十分危险的，这代表着可能导致的钓鱼攻击。

下面是一个范例:

```
pragma solidity >0.4.24;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE
contract TxUserWallet {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function transferTo(address dest, uint amount) public {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

我们可以构造攻击合约

```
pragma solidity >0.4.24;

interface TxUserWallet {
    function transferTo(address dest, uint amount) external;
}

contract TxAttackWallet {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function() external {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

当用户被欺骗调用攻击合约，则会直接绕过鉴权而转账成功，这里应使用msg.sender来做权限判断。

https://solidity.readthedocs.io/en/develop/security-considerations.html#tx-origin

(10) 条件竞争问题

合约中尽量避免对交易顺序的依赖

在智能合约中，经常容易出现对交易顺序的依赖，如占山为王规则、或最后一个赢家规则。都是对交易顺序有比较强的依赖的设计规则，但以太坊本身的底层规则是基于矿工利益最大法则，在一定程度的极限情况下，只要攻击者付出足够的代价，他就可以一定程度控制交易的顺序。开发者应避免这个问题。

真实世界事件

Fomo3d事件

- 智能合约游戏之殇——类 Fomo3D 攻击分析

(11) 未初始化的储存指针

避免在函数中初始化struct变量

在solidity中允许一个特殊的数据结构为struct结构体，而函数内的局部变量默认使用storage或memory储存。

而存在storage(存储器)和memory(内存)是两个不同的概念，solidity允许指针指向一个未初始化的引用，而未初始化的局部stroage会导致变量指向其他储存变量，导致变量覆盖，甚至其他更严重的后果。

```
pragma solidity ^0.4.0;

contract Test {

        address public owner;
        address public a;
```

```
        struct Seed {
                address x;
                uint256 y;
        }

        function Test() {
                owner = msg.sender;
                a = 0x1111111111111111111111111111111111111111;
        }

        function fake_foo(uint256 n) public {
                Seed s;
                s.x = msg.sender;
                s.y = n;
        }
}
```

上面代码编译后，s.x和s.y会错误的指向ownner和a。

攻击者在执行`fake_foo`之后，会将owner修改为自己。

上述问题在最新版的0.4.25版本被修复。

(12)「输入假名」问题

在引入零知识证明作为校验合约的条件时，需要谨防“输入假名”问题。

在密码学中，椭圆曲线的`{x, y}`的值域是一个基于mod p的有限域，基于椭圆曲线的加密也就在这个有限域的循环中进行。在智能合约的实现中，可能存在多个数经过mod运算后都会对应同一个Fq，在这种情况下，我们把这多个大整数称为“输入假名”，而这些数互为假名。这也就意味着，互为假名的多个数可以重复的通过校验，利用5个“输入假名”可以将一笔钱重复花费5次。

真实世界事件

- zkSNARK 合约「输入假名」漏洞致众多混币项目爆雷

以太坊合约审计checkList审计系列报告

- 《以太坊合约审计 CheckList 之“以太坊智能合约编码隐患”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约规范问题”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约设计缺陷问题”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约编码安全问题”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约编码设计问题”影响分析报告》

REF

- https://github.com/ConsenSys/smart-contract-best-practices/blob/master/README-zh.md
- https://dasp.co
- https://etherscan.io/solcbuginfo
- https://www.kingoftheether.com/contract-safety-checklist.html
- https://mp.weixin.qq.com/s/UXK8-ZN7mSUI3mPq2SC6Og
- https://mp.weixin.qq.com/s/kEGbx-I17kzm7bTgu-Nh2g

- https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/Bai%20Zheng%20and%20Chai%20Wang/DEFCON-26-Bai-Zheng-Chai-Wang-You-May-Have-Paid-more-than-You-Imagine.pdf
- https://mp.weixin.qq.com/s/SfKh7_xh7OwV9b31T4t-PQ
- http://rickgray.me/2018/05/17/ethereum-smart-contracts-vulnerabilites-review/
- http://rickgray.me/2018/05/26/ethereum-smart-contracts-vulnerabilities-review-part2/