1. 地址变量address自带的call函数解释

为了同一些不支持ABI协议的进行直接交互（一般的web3.js，soldity都是支持的）。可以使用call()函数，用来向另一个合约发送原始数据。参数支持任何类型任意数量。每个参数会按规则(规则是按ABI[4](http://www.tryblockchain.org/Solidity-Type-Address-%E5%9C%B0%E5%9D%80.html" \l "fn4))打包成32字节并一一拼接到一起。

call()是一个底层的接口，用来向一个合约发送消息[[1]](http://www.jianshu.com/p/fd5075ff0ab9" \l "fn_lemma_call)，也就是说如果你想实现自己的消息传递，可以使用这个函数。函数支持传入任意类型的任意参数，并将参数打包成32字节，相互拼接后向合约发送这段数据。

1. 回退函数fallback的解释

每一个合约有且仅有一个没有名字的函数。这个函数无参数，也无返回值。它在调用合约时，但没有匹配上任何一个函数(或者说没有提供任何的数据)。

此外，当合约收到ether时（没有数据），这个函数也会被执行。在此时，一般仅有少量的gas剩余，用于执行这个函数(准确的说，还剩2300gas)。所以需要尽量保证回退函数尽可能的减少gas花费。

如果是合约地址，在执行send()时，会默认关联执行fallback()（如果存在这个函数）。这是EVM的默认行为，不可被阻止。所以这个函数引起out of gas或其它失败，整个交易被撤销。由于失败，此时的send()的结果是false。

下述提供给回退函数可执行的操作会比常规的花费得多一点。

* 写入到存储(storage)
* 创建一个合约
* 执行一个外部(external)函数调用，会花费非常多的gas
* 发送ether

请在部署合约到网络前，保证透彻的测试你的回退函数，来保证函数执行的花费控制在2300gas以内。

当我们调用某个智能合约时，如果指定的函数找不到，或者根本就没指定调用哪个函数（如发送ether）时，fallback函数就会被调用。

fallback函数被设计成不能做太多事，合理的做法是在fallback函数里打印一些log（或称event），以通知客户端（通常是web3.js）一些相关信息。所以你如果给该调用不发送任何气（就像send()一样），系统会默认给予2300气的上限来执行fallback函数。

但是当你通过addr.call.value()()的方式发送ether，情况就不一样了。和send()一样，fallback函数会被调用，但是传递给fallback函数可用的气是当前剩余的所有气（可能会是很多），这时fallback函数可以做的事情就有很多（如写storage、再次调用新的智能合约等等）。一个精心设计的用于做恶的fallback可以做出很多危害系统的事情。

所以避开这个安全漏洞，结论就是：总是用send()来发送ether，而不是用call.value()。

看一个合约例子：

* pragma solidity ^0.4.0;
* contract Person{
* bytes fail;
* function(){
* fail = msg.data;
* }
* function getFail() returns (bytes){
* return fail;
* }
* }
* contract CallTest{
* function callData(address addr) returns (bool){
* return addr.call("abc", 256);
* }
* }

再来看一个例子：

* pragma solidity ^0.4.0;
* contract Person{
* uint age = 10;
* function increaseAge(string name, uint num) returns (uint){
* return ++age;
* }
* function getAge() returns (uint){
* return age;
* }
* }
* contract CallTest{
* function callByFun(address addr)returns (bool){
* bytes4 methodId = bytes4(keccak256("increaseAge(string,uint256)"));
* return addr.call(methodId,"jack", 1);
* }
* }

3.安全漏洞二：递归调用（recursive）

看下面的代码：

function withdrawBalance() {

amountToWithdraw = userBalances[msg.sender];

if（amountToWithdraw > 0){

if (!(msg.sender.call.value(amountToWithdraw)())) { throw; }

userBalances[msg.sender] = 0;

}

}

这是一段给用户取款的代码，让用户一次性从你的智能合约里取回存款。例如，你的合约账户共有1000个ether，而某用户存有10个ether。因为该代码存有严重的递归调用漏洞，该用户可轻松地将你账户里的1000个ether全部提走。

首先，该段代码使用了addr.call.value()()来发送ether，而不是send()，给fallback函数的调用提供了足够多的gas。你只要将fallback函数写成如下的方式便可取走所有的ether：

function () {

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(COUNT<100){

addr.call("withdrawBalance");

COUNT++;

}

}

在这段fallback代码中，当计数器小于100时，递归调用withdrawBalance函数。在这种情况下，

msg.sender.call.value(amountToWithdraw)()

将被调用100次，从而取走100\*10 ether。

所以在写智能合约时，需要考虑到它可能被递归调用，在这个case里，我们可以这样调整代码以防止递归调用而出现的问题：

function withdrawBalance() {

amountToWithdraw = userBalances[msg.sender];

userBalances[msg.sender] = 0;

if（amountToWithdraw > 0){

if (!(msg.sender.call.value(amountToWithdraw)())) {

userBalances[msg.sender] = amountToWithdraw;

throw;

}

}

}

4.安全漏洞三：调用深度限制（call depth）

调用深度（call depth）被限制为1024。EVM中一个智能合约可以通过message call调用其它智能合约，被调用的智能合约可以继续通过message call再调用其它合约，甚至是再调用回来（recursive）。嵌套调用的深度被限定为1024。

看下面这段代码：

function sendether(){

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

addr.send(20 ether);

//you think the send should return true

var thesendok = true;

//do something regarding send returns ok

...

}

并且对方的fallback函数定义为：

function(){

//do nothing

}

你认为你的代码肯定是安全的，因为对方已经明确定义了fallback方法。但是你错了，攻击者只需要制造出1023个嵌套调用，然后再调用sendether()，就可以让add.send(20 ether)失败，而其它执行成功。代码如下：

function hack(){

var count = 0;

while(count < 1023){

this.hack();//this keyword makes it a message call

count++;

}

if(count==1023){

thecallingaddr.call("sendether");

}

}

所以为了解决深度限制的问题，正确的写法应该是在每次涉及到call depth增加的地方都检查调用返回是否正确，如下：

function sendether(){

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(!addr.send(20 ether)){

throw; //somebody hacks me

}

//you think the send should return true

var thesendok = true;

//do something regarding send returns ok

...

}

了解了智能合约里面的这三个基本的漏洞，下面我们来看看在著名的The DAO事件中，黑客是怎么利用它们来窃取以太币，造成以太坊发展史上影响最为惨重的一次事件。

5.DAO漏洞

The DAO的漏洞是上面第一二个漏洞的组合。看下面的代码：

function splitDAO(

uint \_proposalID,

address \_newCurator)noEther onlyTokenholders returns (bool \_success){

...

uint fundsToBeMoved =

(balances[msg.sender] \* p.splitData[0].splitBalance) /

p.splitData[0].totalSupply;

if(p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)

== false) throw;

...

withdrawRewardFor(msg.sender);

totalSupply -= balances[msg.sender];

balances[msg.sender] = 0;

paidOut[msg.sender] = 0;

return true;

}

黑客通过将下面的代码调用多次，以转移多份以太币：

p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)

他是怎么做到的呢？很简单！当合约执行到：

withdrawRewardFor(msg.sender);

时候，会进入相应的函数：

function withdrawRewardFor(address \_account)

noEther internal returns (bool \_success){

...

if(!rewardAccount.payOut(\_account,reward)) //漏洞代码

throw;

...

}

payOut函数定义如下：

function payOut(address \_recipient, uint \_amount) returns (bool){

...

if(\_recipient.call.value(\_amount)) //漏洞代码

PayOut(\_recipient, \_amount);

return true;

}else{

return false;

}

}

看到这里，你也许已经懂了，这和我们前面给出的例子几乎是一模一样的。代码中通过addr.call.value()()的方式发送以太币，而不是send()，这给黑客留下了空间。黑客只需要制造出一个fallback函数，在该函数里再次调用splitDAO()即可。

6.两种发送货币的方式

先看下面两段代码：

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(!addr.call.value(20 ether)()){

throw;

}

以及：

address addr = 0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b;

if(!addr.send(20 ether)){

throw;

}

这两段代码都是向0x6c8f…的合约地址发送20个ether，第二段代码没有漏洞，而第一段代码却存在严重的安全漏洞。为什么？

我们先来看一下addr.call.value()()（注意：是两个括号，第一个括号是对要转移多少以太币的赋值，第二个括号是方法的调用）和addr.send()的区别。两者都是向某个地址发送以太币，都是一个新的message call，不同的是这两个调用的gaslimit不一样。send()给予0的gas（相当于call.gas(0).value()())，而call.value()()给予全部（当前剩余）的gas。

注：对于需要调用fallback函数又没有给予任何gas的情况，EVM将自动把gas调整为不超过2300。