CVE-2018-12067及类似漏洞分析与相关思考

Pinging / 2019-05-29 08:49:00 / 浏览数 4943 安全技术 区块链安全 顶(0) 踩(0)

一、前言

最近审计了一波以太坊CVE，本章中提及的漏洞包含了众多问题，不仅包括代码上的漏洞，还包括由于函数设计问题而导致的金融学上的漏洞。在分析漏洞之余，我也对本文

该漏洞仅为这一类漏洞的代表，与其相关的类似合约还有许多，这里仅使用最有代表性一个来进行分析演示。

首先需要简单的介绍一下相关背景。本文安全隐患是以PolyAI 、Substratum为代表的一类项目。Substratum项目主要是打造基于区块链的网络，
简而言之，就是重构现在的互联网服务，包括 DNS， 网络空间存储等，其中解决"国际性网络封锁"
，完全为中国定制，可以解决国内需要通过VPN服务才可以翻墙访问国外网站的问题。从官网看来，开发者应该很了解中国国情，网站和白皮书的中文都下了很多功夫，做得

POLY AI 聚焦于基于深度学习的人工智能，谷歌、苹果、Facebook最近在这个领域贡献了巨大的成果，POLY AI
使用区块链技术来解决深度学习训练过程中的计算消耗和数据源问题，而且区块链也比所有传统技术更能够提供安全保障。

这些币种的实用性很强，均为某个领域提供支持。

二、问题所在

该类型的应用包括三类问题。首先为可超额铸币。

为了方便管理员对应用进行管理，设计者在设计此类应用的时候添加了发币接口。此类发币接口的使用者为owner并且能够没有限制的增加自身发币数量。

```
function mintToken(address target, uint256 mintedAmount) onlyOwner {
       balanceOf[target] += mintedAmount;
       totalSupply += mintedAmount;
       Transfer(0, this, mintedAmount);
       Transfer(this, target, mintedAmount);
   }
```

该函数拥有两个参数，包括目标地址与发币数量。当owner调用此函数的时候，其可传入待处理钱包地址并能够任意为钱包账户增加余额。这种方法看似增加了合约的灵活性
如果恶意owner存在，那么他便能够任意增加账户货币数量，当其数量增多时，货币的市场价值便会受到影响。

其次，该类型合约均存在严重的漏洞，即乘法溢出。

下面我将举例两个例子。

首先我们来看Substratum合约中的函数。

在合约的关键转账函数中，设计者对溢出做出了判断。

```
function transfer(address _to, uint256 _value) {
       if (balanceOf[msg.sender] < _value) throw;           // Check if the sender has enough
       if (balanceOf[_to] + _value < balanceOf[_to]) throw; // Check for overflows
       if (frozenAccount[msg.sender]) throw;                // Check if frozen
       balanceOf[msg.sender] -= _value;                     // Subtract from the sender
       balanceOf[_to] += _value;                            // Add the same to the recipient
       Transfer(msg.sender, _to, _value);                   // Notify anyone listening that this transfer took place
   }
```

即：`if (balanceOf[_to] + _value < balanceOf[_to]) throw;`当转账金额溢出时执行throw操作。

然而在后面的卖出函数中却没有进行溢出判断。

```
function mintToken(address target, uint256 mintedAmount) onlyOwner {
       balanceOf[target] += mintedAmount;
       totalSupply += mintedAmount;
       Transfer(0, this, mintedAmount);
       Transfer(this, target, mintedAmount);
   }

   function freezeAccount(address target, bool freeze) onlyOwner {
       frozenAccount[target] = freeze;
       FrozenFunds(target, freeze);
   }

   function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner {
```

```
        sellPrice = newSellPrice;
        buyPrice = newBuyPrice;
    }

    function buy() payable {
        uint amount = msg.value / buyPrice;               // calculates the amount
        if (balanceOf[this] < amount) throw;              // checks if it has enough to sell
        balanceOf[msg.sender] += amount;                  // adds the amount to buyer's balance
        balanceOf[this] -= amount;                        // subtracts amount from seller's balance
        Transfer(this, msg.sender, amount);               // execute an event reflecting the change
    }

    function sell(uint256 amount) {
        if (balanceOf[msg.sender] < amount ) throw;       // checks if the sender has enough to sell
        balanceOf[this] += amount;                        // adds the amount to owner's balance
        balanceOf[msg.sender] -= amount;                  // subtracts the amount from seller's balance
        if (!msg.sender.send(amount * sellPrice)) {       // sends ether to the seller. It's important
            throw;                                        // to do this last to avoid recursion attacks
        } else {
            Transfer(msg.sender, this, amount);           // executes an event reflecting on the change
        }
    }
}
```

在上文函数中，我们能够看到用户能够执行buy()函数并传入一定金额，之后通过owner预设的buyprice转换为对应的具体代币额度，之后增加用户余额。而在sell函数中
*
sellPrice)时出现了问题，由于amount与sellPrice是可设置的，且合约并没有对其进行溢出检测，所以此处存在非常严重的漏洞。倘若owner为作恶节点，或者sellPrice
*
sellPrice就会增加到一个非常大的数额，当此数额庞大到超出uint256限制后，变会产生溢出，此时对于用户来说，其花费了巨大的代价却没有得到回报，是十分不友好的

相比较来说，用户同样可以利用类似的漏洞进行作恶。例如SEC代币：

```
function buy() payable returns (uint256 amount){
        if(!usersCanTrade && !canTrade[msg.sender]) revert();
        amount = msg.value * buyPrice;                    // calculates the amount

        require(balanceOf[this] >= amount);               // checks if it has enough to sell
        balanceOf[msg.sender] += amount;                  // adds the amount to buyer's balance
        balanceOf[this] -= amount;                        // subtracts amount from seller's balance
        Transfer(this, msg.sender, amount);               // execute an event reflecting the change
        return amount;                                    // ends function and returns
    }

    //user is selling us grx, we are selling eth to the user
    function sell(uint256 amount) returns (uint revenue){
        require(!frozen[msg.sender]);
        if(!usersCanTrade && !canTrade[msg.sender]) {
            require(minBalanceForAccounts > amount/sellPrice);
        }
        require(balanceOf[msg.sender] >= amount);         // checks if the sender has enough to sell
        balanceOf[this] += amount;                        // adds the amount to owner's balance
        balanceOf[msg.sender] -= amount;                  // subtracts the amount from seller's balance
        revenue = amount / sellPrice;
        require(msg.sender.send(revenue));                // sends ether to the seller: it's important to do this last to preve
        Transfer(msg.sender, this, amount);               // executes an event reflecting on the change
        return revenue;                                   // ends function and returns
    }
```

我们能够看到，在此合约中同样存在类似的乘法溢出问题。在buy()函数中，我们能发现合约对用户的合法性进行了提前的判断，并计算出购买金额。而此处的购买金额使用
= msg.value * buyPrice。即传入代币数量*购买金额。当用户需要执行大量购买的请求时，amount变会出现溢出从而用高金额获取到低代币数量。

具体的操作流程我们将在下章进行演示。

除了存在乘法溢出漏洞以外，该合约还存在不合理的函数设计，例如合约中均存在设置代币购买卖出的金额。

```
function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner {
        sellPrice = newSellPrice;
        buyPrice = newBuyPrice;
    }
```

例如上述代码中允许owner进行对代币购买价格、卖出价格进行设置。而我们知道，对于代币来说，货币的价格需要通过市场来进行定义，而很大程度需要避免人为设定。

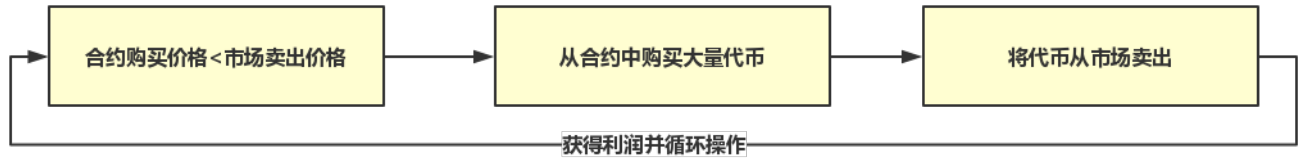
而在合约中我们发现用户购买、卖出代币均有相应的函数，并且此函数均使用先前owner定义的价格。这样将会出现非常大的问题。首先我们会假定owner作恶的情况，当c

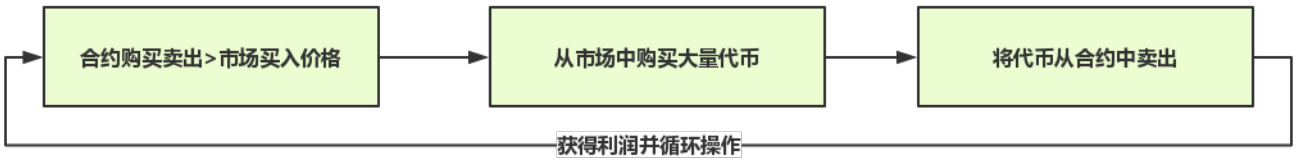
我们知道以太坊应用的代币在市场上同样存在相对应的兑换汇率，即以太币兑换相应Token。然而此处又多出来了第二种兑换方式。这也就导致了中间存在哪一种兑换更合适

1 当合约卖出>市场购买价格时，我们可以在市场上进行购买，并在合约中进行卖出。

2 当合约购买价格<市场卖出价格时，我们通过合约进行对token进行购买并将此token卖出到市场上。

于是我们可以使用以上两种方法进行循环套利。具体方法如图所示：

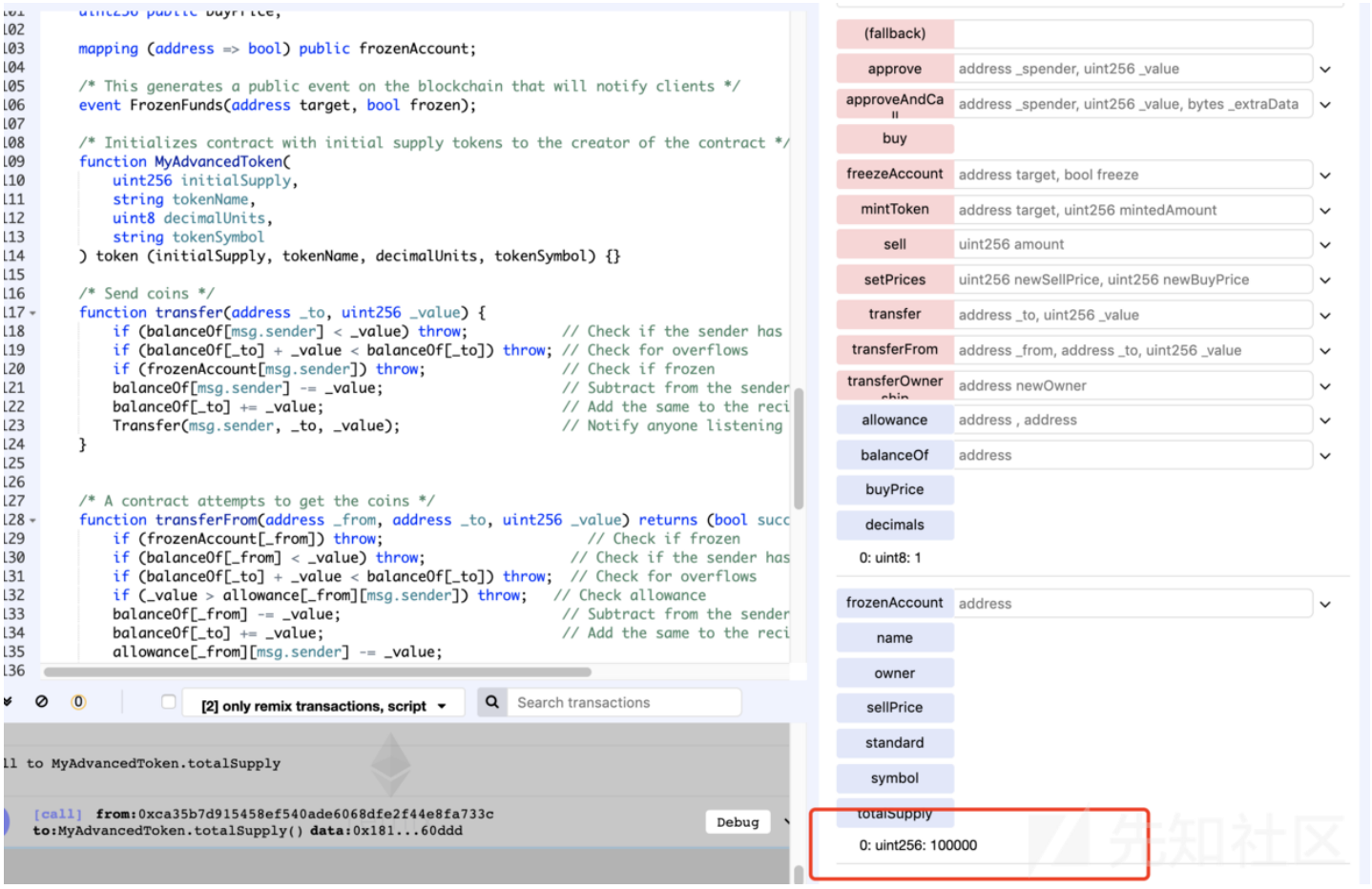




三、漏洞模拟

我们将在本章模拟溢出过程。本次模拟我们使用测试环境并使用两个钱包地址进行交互测试。首先我们部署合约：

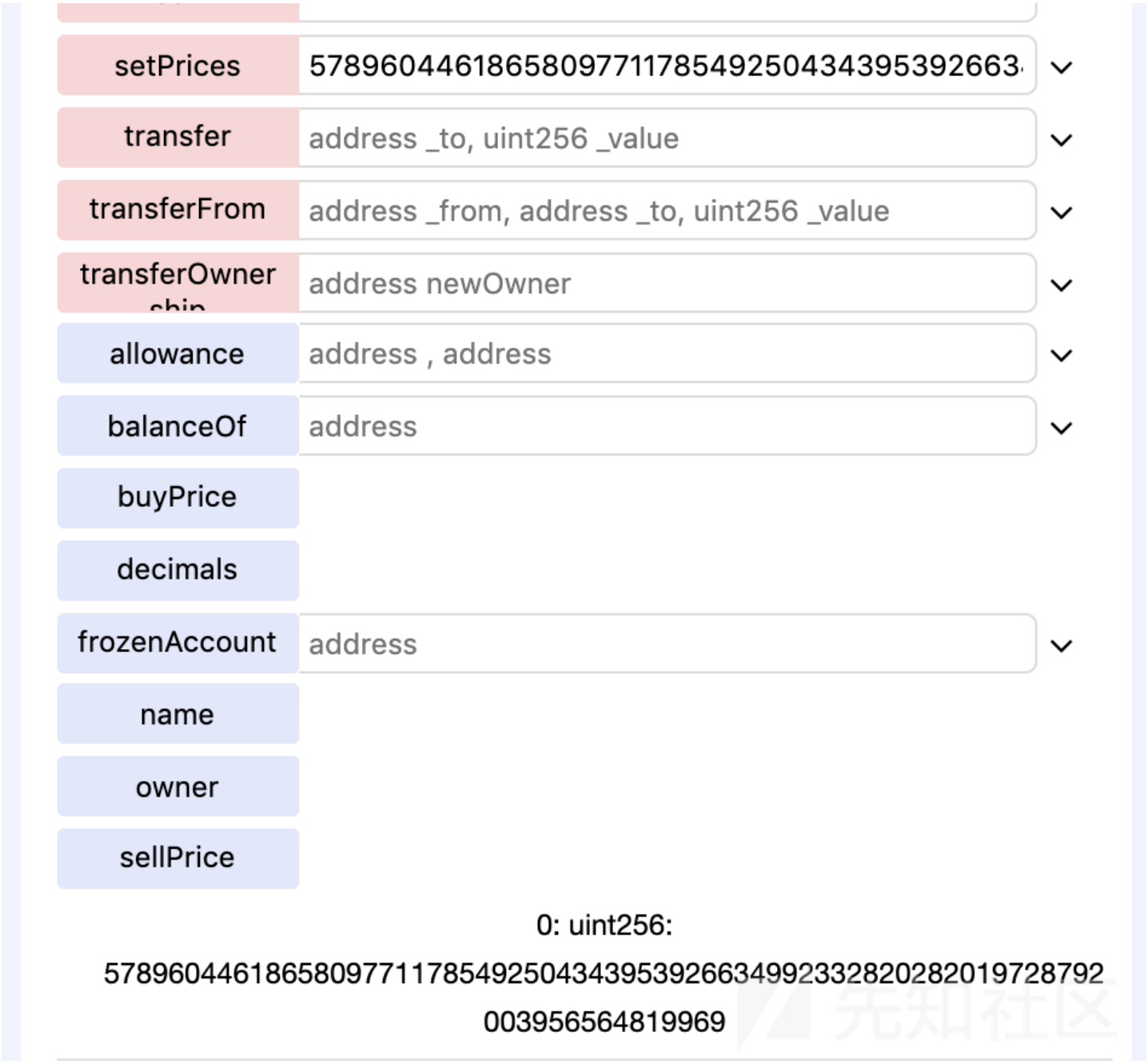
我们使用测试环境且地址为`0xca35b7d915458ef540ade6068dfe2f44e8fa733c`的账户进行。 起始赋值给msg.sender账户余额为100000 。



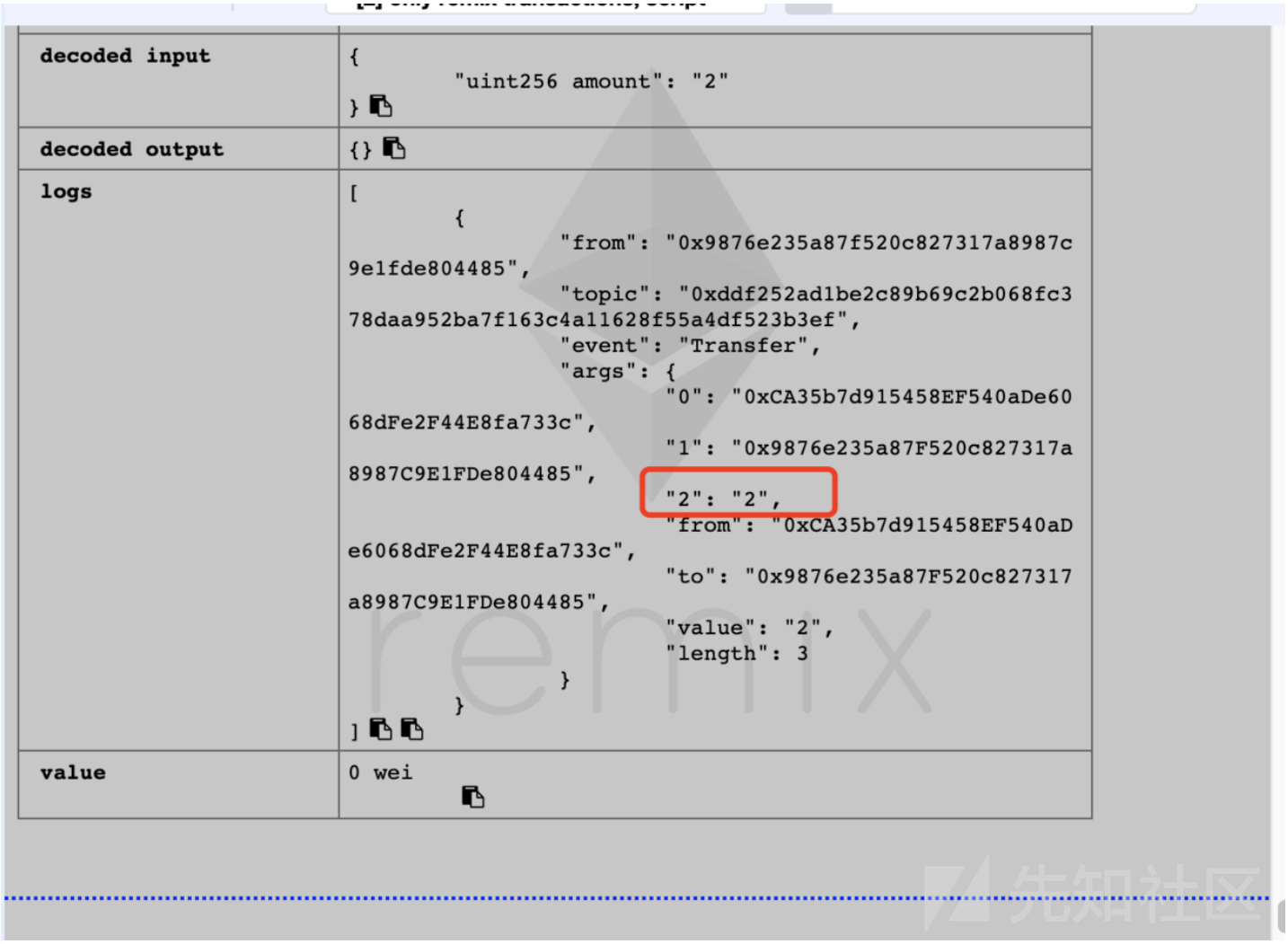
之后我们模拟作恶owner，尝试修改各种代币参数。

首先我们设置代币购买金额为100 。卖出金额为57896044618658097711785492504343953926634992332820282019728792003956564819969
。为什么设置这个数字呢？我们通过计算0xfff.....fff(32个)为115792089237316195423570985008687907853269984665640564039457584007913129639935，而11
= 57896044618658097711785492504343953926634992332820282019728792003956564819967。所以我们设置末尾数为9方便我们查看。

我们能够看到我们这里取了极端情况后，sellPrice如图：



之后我们调用sell函数卖出2个token，由于缺乏溢出检查，所以这里理所当然会产生溢出问题。于是我们可以查看log日志记录。



当卖出2token时，我们应该获取2*57896044618658097711785492504343953926634992332820282019728792003956564819969的以太币，然而这里却只得到了wei的以太币。从而令用户蒙冤。

同样，本溢出适用于下列代码，测试同样可以达成效果。这里就不再过多演示。

```
function buy() payable returns (uint256 amount){
       if(!usersCanTrade && !canTrade[msg.sender]) revert();
       amount = msg.value * buyPrice;                    // calculates the amount

       require(balanceOf[this] >= amount);               // checks if it has enough to sell
       balanceOf[msg.sender] += amount;                  // adds the amount to buyer's balance
       balanceOf[this] -= amount;                        // subtracts amount from seller's balance
       Transfer(this, msg.sender, amount);               // execute an event reflecting the change
       return amount;                                    // ends function and returns
   }
```

本合约存在代码层面以及设计方面的问题，目前来看隐患并不是很大，但是存在了犯错误的可能。由于区块链信奉"代码即法律"的理念，所以不应当出现这种问题隐患。

四、参考资料

- https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12082

  https://blog.peckshield.com/2018/06/11/tradeTrap/

  https://etherscan.io/address/0x5121e348e897daef1eef23959ab290e5557cf274

  https://etherscan.io/address/0x12480e24eb5bec1a9d4369cab6a80cad3c0a377a

本稿为原创稿件，转载请标明出处。谢谢。

上一篇：强网杯线上赛 题解 - Lilac 下一篇：CVE-2019-1003031:...

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板