合约漏洞整理分析

https://www.chaincatcher.com/article/2059876

智能合约漏洞：4大分类共39种情况

4大分类：以太坊应用层漏洞、以太坊数据层漏洞、以太坊共识层漏洞、以太坊网络层漏洞

应用层

重入攻击（Re-Entrancy）

就是合约A通过函数funcA调用攻击合约B中的一个函数funcB，B中的该函数funcB又返回来调用funcA，从而达到攻击效果。

比如合约A调用函数funcA，funcA中通过`B.call.value(_money)`给B合约转账，该call调用会执行B合约中的回调函数，如果B合约中的回调函数又反过来调用funcA，那么A合约账户就会重复给B合约账户转账。

在著名的The DAO事件中，黑客就使用了这种攻击。

解决方法是：先更改合约状态变量值，再执行真正的转账操作。比如先更改余额，再转账。

DelegateCall 攻击

使用delegateCall调用其他合约的函数时，代码是在调用合约的环境里执行，内置变量`msg`的值不会修改为`调用者`，但执行环境为`调用者`的运行环境，即使是`private`类型的变量，也会被delegatecall调用更改。比如：

```
pragma solidity ^0.8.0;

contract A {
    address public temp1;
    uint256 public temp2;

    function three_call(address addr) public {
        addr.delegatecall(bytes4(keccak256("test()")));
    }
}

contract B {
    address public temp1;
    uint256 public temp2;

    function test() public  {
        temp1 = msg.sender;
        temp2 = 100;
    }
}
```

调用合约函数的外部账户地址为0xca35b7d915458ef540ade6068dfe2f44e8fa733c，调用函数three_call，观察变量的值发现合约 B 中变量值仍然为 `0x0`，而**调用者**合约 A 中的 `temp1 = 0xca35b7d915458ef540ade6068dfe2f44e8fa733c, temp2 = 100`.

这种情况，被调用合约有机会恶意的修改（或操纵）调用方合约的状态变量。

Parity钱包最先被发现遭遇此攻击。

解决方法：通过delegatecall调用无状态合约，即可完全防止该漏洞。

冻结Ether 攻击

漏洞使得黑客能通过库函数成为库的主人，然后调用自杀函数 `selfdestruct` 报废整个合约库。使得依赖该库的钱包提款功能都失效，150多个地址中超过50万个ETH被彻底冻结。

Parity钱包最先被发现遭遇此攻击。

解决方法：注意合约及函数的owner问题，以及谨慎使用自杀函数。

升级合约攻击

引入合约升级的思想是为了缓解智能合约一旦部署后就无法修改的问题（即使以后发现它们存在漏洞，也无法修改）。为了允许合约升级，一般都是：将合约梳理为代理合约和逻辑合约，从而

可以升级逻辑合约；使用注册管理机构合约来保存更新后的合约。这些方法虽有效，但却引入了新的漏洞：若合约开发者变得恶意时，更新的合约可能是恶意的。

解决方法：升级逻辑合约的权限分散到5个账户上，获得至少3个账户的授权才能升级合约，5个账户分别由不同人员管理。

带有意外revert的DoS 攻击

攻击者故意触发被攻击合约的revert，导致交易被还原，中断交易的执行。

解决方法：接收者调用交易来'提取'发送者为接收者预留的资金。这样就有效地防止了发送者的交易被还原。实际写合约时，需要注意被恶意利用的revert攻击即可。

整数上溢和下溢 攻击

运算的结果超出了Solidity数据类型的范围，就会出现上溢和下溢的问题，从而改变数据。比如对攻击者的余额进行未经授权的操作。

解决方法：在Solidity 0.8.0版本之后，所有的上溢或下溢操作都会默认的出现revert错误而导致交易失败、执行回退，就不需要额外引入SafeMath类似的库了。

tx.origin 攻击

外部账户EOA调用合约A的transferTo函数，给攻击合约B转账，transferTo函数触发了合约B的`receive`函数，而攻击合约的receive函数又返回来调用了transferTo函数，从而给自己多次转账，因为此时transferTo函数里面判断tx.origin仍然还是这个外部账户。

被攻击的代码如下：

```
contract TxUserWallet is DSTest {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint256 amount) public payable {
        emit log_named_address("tx.origin", tx.origin);
        emit log_named_address("owner", owner);
        require(tx.origin == owner); //** 重点 **
        dest.call{value: amount}("");
    }
}
```

`TxUserWallet` 合约调用transferTo 给攻击合约转账，攻击合约在 `receive` 函数中再次调用transferTo，这个时候，tx.origin 就还是原来的账号操作人，代码写得好，攻击合约就可以转空 `TxUserWallet` 合约里的钱。

解决方法：

错误的可见性

一些重要函数指定了错误的可见性，从而允许未经授权的访问。

解决方法：仔细检查所写的代码。

无保护的自杀

合约所有者可以使用自杀'selfdestruct'函数销毁合约，销毁合约时，将删除其关联的字节码和存储。该漏洞是由合约强制执行的身份验证不足引起的，导致攻击者恶意调用合约自杀函数。

解决方法：自杀操作必须得到多方的批准。

以太币泄漏到任意地址

缺乏身份验证，从而导致可以将以太币发送到任意地址。

解决方法：需要对发送资金的功能进行适当的身份验证。

机密性攻击

要注意，由于区块链的公共性质（即交易细节是众所周知的），限制变量或函数的可见性并不能确保变量或函数是机密的，只能确保不能被其他合约调用访问。

解决方法：不要在合约中存储机密信息。

重放攻击

当数字签名对多个交易有效时，就会发生此漏洞。即一条签名信息，可以用来多次触发交易。

解决方法：在每条签名消息中合并适当的信息，比如时间戳等。

带有超出gaslimit的DoS 攻击

通过无限扩大合约中的数组元素个数，从而导致合约中包含for循环遍历数组的函数总是执行失败，来攻击合约。

解决方法：注意合约中的for循环操作，使其不至于超出区块gaslimit。

unchecked 漏洞

合约里面当使用unchecked时，就不会检查unchecked模块中调用的返回值和出现的错误。模块外的操作会继续执行。

注意：谨慎使用unchecked

未初始化的存储指针

在solidity中，动态的数组、struct、mapping这样的复杂数据结构是不能直接存储在"栈"中的（即卡槽），因为"栈"只能保存实际数据长度小于等于32字节的简单数据类型。所以声明动态数组、struct时，需指定storage或memory。在函数内部，mapping不能作为临时变量，只能作为某个状态变量的"存储指针"，也就是mapping类型必须在编译时进行预先初始化，而不能作为运行时产生的数据。如果智能合约函数声明了临时的动态数组或者struct，而没有指定"位置"（storage或memory），那么这些变量将默认为"存储指针"。比如：

```
contract PledgeAddT {
    bool public unlocked = false;
    struct NameRecord {
        bytes32 name;
        address mappedAddress;
    }
    mapping(address=>NameRecord) public reg;

    function register(bytes32 _name, address _mappedAddress) public {
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;
        reg[msg.sender] = newRecord;
        require(unlocked);
    }
}
```

这个合约声明了2个状态变量：unlocked、reg，它们分别会存储在存储槽0、1的位置。register函数第一行声明了struct类型的变量，但没有指明"位置"，所以它会被默认为"存储指针"。程序中还没有对其进行初始化，所以它将指向存储槽0，即状态变量unlocked的位置。这样，当后面对newRecord.name赋值时，实际修改的就是状态变量unlocked的值。

注意：在solidity0.8.0 版本之后，编译器会强制让合约编写者给函数内的临时复杂变量指定storage、memory或calldata，否则就会报错。

错误的构造函数名称

导致任何人都可以成为合约的所有者。

注意：在solidity0.4.22版本后，构造函数名称统一为`constructor`.

合约地址类型转换

可以通过直接引用被调用方合约的实例来调用另一个合约。将合约地址转换为合约类型，比如：

```
IGroupSetter(insti.instances(11)).setP(_gi, _kp, _pp);
```

但是合约编译时并不会检查`insti.instances(11)`获得的合约地址是否是GroupSetter合约的地址。

注意：编写部署合约后，要进行基本的测试；部署合约时，也要谨慎细心。

使用过时的编译器版本

使用过时的编译器可能会包含错误，并且容易受到攻击。

注意：尽量使用最新的编译器。

短地址攻击

比如一个用户在交易所的钱包上（钱包余额1000tokens）有一个余额32tokens，假设用户的地址是0x12345600（为了方便，少写了一些0。即地址是以0结尾的），现在用户想提取超出他余额的资金，他点击交易所的取款按钮，并且输入自己的地址为0x123456，故意少输入最后的0。而交易所并没有验证地址的长度，直接将输入的数据编码成交易信息发送了出去，并且交易执行成功。然后，EVM计算输入的交易data，根据地址应有的长度来解析地址，amount编码高位补0，导致amount编译值比实际输入值大，从而进行错误的转账。ERC20的转账格式为：`transfer(address to, uint256 amount)`，这3个参数如下：

```
function signature: a9059cbb = web3.sha3("transfer(address,uint256)").slice(0,10)
arg1: 123456   = receiving address
arg2: 00000020  = 32 in hexademical (0x20)
----------------------------------------
Concatenated: a9059cbb 123456 00000020
Transaction input data: 0xa9059cbb12345600000020
```

解决方法：在交易发起端，需验证输入数据格式的正确性，特别是地址长度的正确性。

以太币等丢失给孤立地址

在转账汇款时，以太坊并不会检查地址的有效性，只会检查是否是20字节。如果转账给一个孤立地址，该地址不属于任何EOA账户（外部账户）、或者转给不能取出来的合约账户，比如地址

0x0，那么该笔资金就无法再使用。

解决方法：手动检测转账账户的正确性，接收资金的合约账户要留有转出资金的接口。

调用堆栈深度攻击

EVM调用堆栈深度限制为1024，因此，外部函数调用随时可能失败，因为它们超过了最大调用堆栈大小限制1024. （EVM是堆栈机，所有计算都在成为堆栈的数据区域上执行，它的最大大小为1024个元素并且每个元素包含256bits）

注意：由于Tangerine Whistle硬分叉，63/64规则使得调用堆栈深度攻击变得不可能。

send转账

如果使用`send`进行转账的话，要注意判断send操作的结果。send操作会返回true或者false，用来表示转账成功或者失败，如果send失败的话，是不会报错的，后面的操作还会继续执行。（不像transfer，transfer失败会造成revert错误，后面的操作不会再执行，前面的操作也会回退）。

低级函数 call、delegatecall 和 staticcall 都和 send 一样，在发生异常时，返回 false，而不是抛出 revert 异常。如果调用的账户不存在，作为 EVM 设计的一部分，低级函数 call、delegatecall 和 staticcall 会返回 true 作为它们的第一个返回值，因此，需要在使用前检查账户是否存在。

注意：使用send转账，则应判断其返回值。

selfdestruct攻击

注意：selfdestruct将合约代币转给目标合约账户时，不会执行目标合约的receive或者fallback函数。

EXTCODESIZE操作码攻击

EXTCODESIZE操作码是用来读取合约的code的大小的，所以其涉及到了相应的磁盘操作，但是它所需的gas又非常少，这就导致了恶意的攻击者可以在交易里实现调用很多次这个操作码，只要这些操作的gas加起来不超过区块的gaslimit即可。当时的攻击交易在每个区块调用了该操作码接近50000次，因此这一个区块内的交易所占用的计算时间就被大大延长（多次访问磁盘，增大了时间消耗），从而导致整个以太坊网络的瘫痪，造成了DDoS攻击（分布式拒绝服务攻击，也可直接称为DoS攻击）。

注意：这种攻击发生在2016年9月份，之后以太坊进行了相应的处理，使得大大降低了类似攻击的风险。官网处理说明。

交易顺序依赖

在并发性的交易中，要注意nonce的取值。区块链的状态变化取决于交易的执行顺序。

时间戳攻击

block.timestamp表示当前区块的时间戳，这个变量通常被用来计算随机数、锁定资金、转移资金的触发条件等。但是区块的打包时间并不是系统设定的，而是可以由矿工在一定程度上进行调整。因此，矿工可能会操纵时间戳生成对自己有利的随机数，从而造成攻击。

以太坊中时间戳的合理要求：

- 当前区块的时间戳一定大于上一个区块的时间戳
- 当前区块的时间戳与上一个区块的时间戳之差小于900s
- 矿工可以在这个"合理"范围内任意设置时间戳

注意：谨慎使用block.timestamp，使其可以容忍900s的时间区间。

随机数攻击

许多赌博和彩票合约都是随机选择中奖者，通常做法是根据一些初始私有种子（例如block.number、block.timestamp、block、difficulty、blockhash）生成伪随机数，但是，这些种子由矿工完全控制，恶意矿工可以操纵这些变量使自己成为赢家。

注意：合约中没有绝对安全的随机数，谨慎使用随机数。

数据层

难以分辨的链

在进行EIP-155硬分叉之前，以太坊每笔交易都包含6个字段（随机数，收件人，值，输入数据，gasPrice，gasLimit），使用ECDSA签署交易，但是，数字签名不是特定于链的。结果，为一个链创建的事务可被另一条链重用。

EIP-155硬分叉之后，通过将chainID合并到字段中，已消除了此漏洞。

空账户攻击

2016年9月份，以太坊网络遭遇了DDoS攻击，由于以太坊协议的漏洞（可以使用SUICIDE操作码廉价地生成新账户），导致攻击者可以在花费很少gas的情况下创建大量的空账户（empty account，balance、nonce、code都为0），空账户功能性上与未存在账户（non-existence account）相似，但是不同点在于空账户会存在于以太坊的 `account state trie` 中。所以这次DDoS攻击导致以太坊节点存储空间爆炸（大概增加了1900万空账户，但是账户与 account state

trie 中的节点不是1：1的关系，因为有中间节点，所以真正增加的存储空间大于1900万账户所需空间）。黄皮书对 SUICIDE 的解释：Halt execution and register account for later deletion.

解决方法：为了解决该漏洞，以太坊进行了硬分叉 `spurious dragon`，旨在清除空账户，减小以太坊节点所需存储空间，修改了协议漏洞（SUICIDE 创建新账户时需要额外的25000 gas），并且引进了新的协议规则：empty account 不再允许被创建（原先可以导致空账户产生的操作将会使账户变为 non-existence）；可以通过交易与空账户交互（touch）来删除它们。

共识层

可外包的PoW

以太坊采用名为 Ethash 的 PoW 难题，该难题旨在抵御 ASIC，并能够限制并行计算的使用。但是，狡猾的矿工仍然可以将搜索难题解决方案的任务划分为多个较小的任务，然后将其外包。

注意：以太坊预计将在2022年第三季度或第四季度转为 PoS 权益证明。

概率最终性

在分布式领域，有个著名的 CAP 定理：分布式系统无法同时确保一致性（Consistency）、可用性（Availability）和分区容忍性（Partition），设计中需要弱化对某个特性的需求。CAP 原理认为，分布式系统最多只能保证三项特性中的两项特性。

以太坊区块链在设计时，更偏向于可用性而非一致性，由此造成会产生分叉的漏洞。

带有块填充的DoS

以太坊约14s左右会出一个块，矿工会优先挑选 gas 消耗比较大、gasPrice 比较高的交易进行打包以便获取更大的利益，目前一个区块的gasLimit 一般为30,000,000. 而对于每一笔交易来说，交易发起者也可以定义一个 gas limit，如果交易消耗的 gas 总值超过 gas limit，该交易就会失败，而大部分交易，会在交易失败时回滚。回滚后，消耗的 gas 就会白白丢失，交易也不会被记录。区块的 gasLimit 决定了区块可以包含的交易数量。

在 solidity 0.8.0以前，指令 `assert()` 允许交易失败但不进行交易回滚操作，所以作恶者完全可以利用这个函数来达到交易无法回滚、从而使交易被矿工打包的情况，而这些恶意的占用了大量 gas 的交易就会将存量不高的区块占领。这就会导致以太坊出块都被这些高 gas 消耗、高 gas limit 的交易包含，从而暂时无法打包其他正常交易。

此漏洞是由贪婪的采矿激励机制引起的。

目前，`assert()` 指令会造成 revert 回退操作。

诚实的挖矿设想

此漏洞是由共识协议引起的，因为它与激励政策不兼容。激励兼容是指：激励策略会激励矿工按照规定遵守协议。实际上，勾结矿工的收入大于他们的公平份额。

出块奖励

出块奖励机制，用于应对由于快速生成块而导致的陈旧区块增加。但是这种机制有一个副作用，即允许自私的矿工将陈旧的区块变成出块并获得奖励，从而有效地激励了自私的采矿和双重支出。

注意：由于目前调整了出块难度，使得自私挖矿很难获益。

验证者的困境

当验证新交易需要不费吹灰之力的计算时，无论矿工是否选择验证交易，都将受到攻击。如果矿工验证了计算量大的交易，那么他们将花费大量时间并在下一个区块的竞争中未攻击者提供优势；如果矿工未经验证接受交易，则区块链可能包含不正确的交易。

注意：可以通过限制验证块中所有事务所需的计算量来缓解此漏洞。

网络层

无限节点创建

该漏洞针对 Geth 客户端1.8之前的版本。攻击者可以在一台计算机上创建无限数量的节点（即具有相同的 IP 地址），然后使用这些节点垄断某些受害者节点的传入和传出连接，从而有效地将受害者和网络中的其他对等节点隔离开来。

不受限制的传入连接

该漏洞针对 Geth 客户端1.8之前的版本。每个节点在任何时间点可以有共 maxpeers 的连接（默认值为25），并且可以与其他节点发起多达（1+maxpeers）/2个出站TCP连接。但是其他节点启动的传入TCP连接数量没有上限，通过为受害节点建立maxpeers量的许多传入连接，攻击者有机会使受害者孤立。

后来，在Geth v1.8中，限制节点的传入TCP连接的数量上限为8，就消除了该漏洞。

节点选择

该漏洞指的是Geth客户端在从其路由表中选择节点以建立出站连接时，总是获取随机选择的桶的头部。由于每个存储桶中的节点都是按活动排序的，因此攻击者可以通过定期向Geth客户端发送

消息来使其节点始终位于其他节点之前。

通过在路由表中所有节点的集合中随机选择统一的节点，而不是仅从每个存储桶的头中随机选择节点，在Geth v1.9中已消除了此漏洞。

RPC API暴露

攻击流程：

- 全局扫描开放8545端口（HTTP JSON RPC API）和8546端口（WebSocket JSON RPC API）等的以太坊节点，发送eth_getBlockByNumber、eth_accounts、eth_getBalance，遍历区块高度、钱包地址、余额。
- 反复调用 eth_sendTransaction 尝试将余额转移到攻击者的钱包。
- 当节点用户碰巧在他的钱包上执行了 unlockAccount 时，他在持续期间不需要重新输入密码来签署交易。此时，攻击者的 eth_sendTransaction 调用将正确执行，余额将转入攻击者的钱包。

使用 JavaScript 控制台时，密码和解锁持续时间都是可选的。如果未将密码短语作为参数提供，控制台将以交互方式提示输入密码短语。

直到解锁持续时间到期，未加密的密钥都将保存在内存中，解锁持续时间默认为 300 秒。如果将解锁持续时间显式设定为零秒，则密钥会保存在内存中直到 geth 退出。

解决方法：

- 启动的geth等客户端节点，更改其默认的 RPC API 端口；
- 配置 iptables 以限制对 RPC API 端口的访问，比如只允许192.168.0.101访问 RPC API 端口；
- 不将 keystore 存储在节点上；
- 使用 web3的sendTransaction和sendRawTransaction发送私钥签名的交易
- 私钥物理隔离（比如冷钱包、人工转录等）或高强度加密存储

应用实例

拍卖

为了最小化外部调用失败带来的损失，通常好的做法是将外部调用函数与其余代码隔离，最终由收款方主动调用函数来进行收款（"pull"版），即让用户自己取款而不是直接发送给他们（"push"版）。

> 记录最高的拍卖价格，当有更高的拍卖价格时，就记录这次的最高拍卖价格和拍卖者，并把原先的拍卖钱退还给上一个拍卖者。

"push"版合约代码（即主动给账户转账的模式）：

```
contract auction {
    address highestBidder;
    uint highestBid;
    function bid() payable {
        if(msg.value<=highestBid) throw;
        if(highestBidder!=address(0)){
            if(!highestBidder.send(highestBid)){
                throw;   // 如果这笔转账一直失败，则会导致其他账户无法进行拍卖
            }
        }
        highestBidder = msg.sender;
        highestBid = mag.value;
    }
}
```

"pull"版合约代码（即账户自己取款的模式）：

```
contract auction {
    address highestBidder;
    uint highestBid;
    mapping(address=>uint) bids;
    function bid() payable external {
        if(msg.value<=highestBid) return;
        if(highestBidder != address(0)){
            bids[highestBidder] += highestBid; //上次的最高竞价者可以取走他的钱
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
    function withdraw() external {
        uint mount = bids[msg.sender];
        if(mount>0){
            bids[msg.sender] = 0;
            if(!msg.sender.send(mount)){
                revert;
            }
        }
    }
}
```