DASP Top 10

这是分布式应用安全项目（或DASP）2018 年排名前10的漏洞第一次迭代

该项目是NCC集团的一项举措。这是一个开放的合作项目，致力于发现安全社区内的智能合约漏洞。要参与，请加入github页面。

1.重入漏洞

也被称为 或与空白竞争，递归调用漏洞，未知调用

这种漏洞在很多时候被很多不同的人忽略：审阅者倾向于一次一个地审查函数，并且假定保护子例程的调用将安全并按预期运行。

菲尔戴安

重入攻击，可能是最着名的以太坊漏洞，第一次被发现时，每个人都感到惊讶。它在数百万美元的抢劫案中首次亮相，导致了以太坊的分叉。当初始执行完成之前，外部合同调用被允许对调用合同进行新的调用时，就会发生重新进入。对于函数来说，这意味着合同状态可能会在执行过程中因为调用不可信合同或使用具有外部地址的低级函数而发生变化。

损失：估计为350万ETH（当时约为5000万美元）

发现时间表：

日期	事件
2016年6月5日	Christian Reitwiessner发现了一个坚定的反模式
2016年6月9日	更多以太坊攻击：Race-To-Empty是真正的交易（vessenes.com）
2016年6月12日	在以太坊智能合约'递归调用'错误发现（blog.slock.it）之后，没有DAO资金面临风险。
2016年6月17日	我认为TheDAO现在正在流失（reddit.com）
2016年8月24日	DAO的历史和经验教训（blog.sock.it）

真实世界影响：

- DAO

示例：

1. 一个聪明的合同跟踪一些外部地址的平衡，并允许用户通过其公共资金检索withdraw()功能。
2. 一个恶意的智能合同使用withdraw()函数检索其全部余额。
3. 在更新恶意合同的余额之前，受害者合同执行call.value(amount)() 低级别函数将以太网发送给恶意合同。
4. 该恶意合同有一个支付fallback()接受资金的功能，然后回调到受害者合同的withdraw()功能。
5. 第二次执行会触发资金转移：请记住，恶意合同的余额尚未从首次提款中更新。结果， 恶意合同第二次成功退出了全部余额。

代码示例：

以下函数包含易受重入攻击影响的函数。当低级别call()函数向msg.sender地址发送ether时，它变得易受攻击; 如果地址是智能合约，则付款将触发其备用功能以及剩余的交易气体：

```
function withdraw(uint _amount) {
        require(balances[msg.sender] >= _amount);
        msg.sender.call.value(_amount)();
        balances[msg.sender] -= _amount;
}
```

其他资源：

- DAO智能合约
- 分析DAO的利用
- 简单的DAO代码示例
- 重入代码示例
- 有人试图利用我们的智能合约中的一个缺陷，盗取它的一切

2.访问控制

通过调用initWallet函数，可以将Parity Wallet库合约变为常规多sig钱包并成为它的所有者。

平价

访问控制问题在所有程序中都很常见，而不仅仅是智能合同。事实上，这是OWASP排名前10位的第5位。人们通常通过其公共或外部功能访问合同的功能。尽管不安全的可视性设置会给攻击者直接访问合同的私有价值或逻辑的方式，但访问控制旁路有时更加微妙。这些漏洞可能发生在合约使用已弃用tx.origin的验证调用者时，长时间处理大型授权逻辑require并delegatecall在代理库或代理合约中鲁莽使用。

损失：估计为150,000 ETH（当时约3000万美元）

真实世界影响：

- 奇偶校验错误1
- 奇偶校验错误2
- Rubixi

示例：

1. 一个聪明的合同指定它初始化它作为合同的业主的地址。这是授予特殊特权的常见模式，例如提取合同资金的能力。
2. 不幸的是，初始化函数可以被任何人调用，即使它已经被调用。允许任何人成为合同的拥有者并获得资金。

代码示例：

在下面的例子中，合约的初始化函数将函数的调用者设置为它的所有者。然而，逻辑与合约的构造函数分离，并且不记录它已经被调用的事实。

```
function initContract() public {
        owner = msg.sender;
}
```

在Parity multi-sig钱包中，这个初始化函数与钱包本身分离并在“库”合同中定义。用户需要通过调用库的函数来初始化自己的钱包delegateCall。不幸的是，在我们的例子中，函数没有检查钱包是否已经被初始化。更糟糕的是，由于 library 是一个聪明的合约，任何人都可以自行初始化 library 并要求销毁。

其他资源：

- 修复Parity多信号钱包bug 1
- 奇偶校验安全警报2
- 在奇偶钱包multi-sig hack上
- 不受保护的功能
- Rubixi的智能合约

3.算术问题

也被称为 整数溢出和整数下溢

溢出情况会导致不正确的结果，特别是如果可能性未被预期，可能会影响程序的可靠性和安全性。

Jules Dourlens

整数溢出和下溢不是一类新的漏洞，但它们在智能合约中尤其危险，其中无符号整数很普遍，大多数开发人员习惯于简单int类型（通常是有符号整数）。如果发生溢出，许多良性代码路径成为盗窃或拒绝服务的载体。

真实世界影响：

- DAO
- BatchOverflow（多个令牌）
- ProxyOverflow（多个令牌）

示例：

1. 一个聪明的合约的withdraw()功能，您可以为您接收eth捐赠，只要你的balance仍然是积极的运算。
2. 一个攻击者试图收回比他或她的当前余额多。
3. 该withdraw()功能检查的结果总是正数，允许攻击者退出超过允许。由此产生的余额下降，并成为比它应该更大的数量级。

代码示例：

最直接的例子是一个不检查整数下溢的函数，允许您撤销无限量的标记：

```
function withdraw(uint _amount) {
        require(balances[msg.sender] - _amount > 0);
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
}
```

第二个例子（在无益的Solidity编码竞赛期间被发现）是由于数组的长度由无符号整数表示的事实促成的错误的错误：

```
function popArrayOfThings() {
        require(arrayOfThings.length >= 0);
        arrayOfThings.length--;
}
```

第三个例子是第一个例子的变体，其中两个无符号整数的算术结果是一个无符号整数：

```
function votes(uint postId, uint upvote, uint downvotes) {
        if (upvote - downvote < 0) {
                deletePost(postId)
        }
}
```

第四个示例提供了即将弃用的var关键字。由于var将自身改变为包含指定值所需的最小类型，因此它将成为uint8保持值0.如果循环的迭代次数超过255次，它将永远达不到该数字，并且在执行运行时停止出气：

```
for (var i = 0; i < somethingLarge; i ++) {
        // ...
}
```

其他资源：

- SafeMath防止溢出
- 整数溢出代码示例

4.未检查返回值的低级别调用

也称为 或与静默失败发送，未经检查发送

尽可能避免使用低级别的“调用”。如果返回值处理不当，它可能会导致意外的行为。

混音

其中的密实度的更深层次的特点是低级别的功能call()，callcode()，delegatecall()和send()。他们在计算错误方面的行为与其他Solidity函数完全不同，因为他们不会传播（或冒泡），并且不会导致当前执行的全部回复。相反，他们会返回一个布尔值设置为false，并且代码将继续运行。这可能会让开发人员感到意外，如果未检查到这种低级别调用的返回值，则可能导致失败打开和其他不想要的结果。请记住，发送可能会失败！

真实世界影响：

- 以太之王
- Etherpot

代码示例：

下面的代码是一个当忘记检查返回值时会出错的例子send()。如果调用用于将ether发送给不接受它们的智能合约（例如，因为它没有应付回退功能），则EVM将用其替换其返回值false。由于在我们的例子中没有检查返回值，因此函数对合同状态的更改不会被恢复，并且etherLeft变量最终会跟踪一个不正确的值：

```
function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
}
```

其他资源：

- 未经检查的外部写入
- 扫描“未经检查 - 发送”错误的现场以太坊合同

5.拒绝服务

包括达到gas上限，意外抛出，意外杀死，访问控制违规

我不小心杀了它。

在Parity多sig钱包上的devops199

在以太坊的世界中，拒绝服务是致命的：尽管其他类型的应用程序最终可以恢复，但智能合同可以通过其中一种攻击永远脱机。许多方面导致拒绝服务，包括在作为交易接受方时恶意行为，人为地增加计算功能所需的气体，滥用访问控制访问智能合约的私人组件，利用混淆和疏忽，...这类攻击包括许多不同的变体，并可能在未来几年看到很多发展。

损失：估计为514,874 ETH（当时约3亿美元）

真实世界影响：

- 政府
- 奇偶校验多信号钱包

示例：

1. 一个拍卖合约允许其用户在竞标不同的资产。
2. 为了投标，用户必须bid(uint object)用期望的以太数来调用函数。拍卖合约将把以太保存在第三方保存中，直到对象的所有者接受投标或初始投标人取消。这意味着拍卖合约必须在其余额中保留未解决出价的全部价值。

3. 该拍卖合约还包含一个withdraw(uint amount)功能，它允许管理员从合约获取资金。随着函数发送amount到硬编码地址，开发人员决定公开该函数。
4. 一个攻击者看到了潜在的攻击和调用功能，指挥所有的合约的资金为其管理员。这破坏了托管承诺并阻止了所有未决出价。
5. 虽然管理员可能会将托管的钱退还给合同，但攻击者可以通过简单地撤回资金继续进行攻击。

代码示例：

在下面的例子中（受以太王的启发），游戏合约的功能可以让你成为总统，如果你公开贿赂前一个。不幸的是，如果前总统是一个聪明的合约，并导致支付逆转，权力的转移将失败，恶意智能合约将永远保持总统。听起来像是对我的独裁：

```
function becomePresident() payable {
   require(msg.value >= price); // must pay the price to become president
   president.transfer(price);   // we pay the previous president
   president = msg.sender;      // we crown the new president
   price = price * 2;           // we double the price to become president
}
```

在第二个例子中，调用者可以决定下一个函数调用将奖励谁。由于for循环中有昂贵的指令，攻击者可能会引入太大的数字来迭代（由于以太坊中的气体阻塞限制），这将有效地阻止函数的功能。

```
function selectNextWinners(uint256 _largestWinner) {
        for(uint256 i = 0; i < largestWinner, i++) {
                // heavy code
        }
        largestWinner = _largestWinner;
}
```

其他资源：

- 奇偶Multisig被黑客入侵。再次
- 关于Parity multi-sig钱包漏洞和Cappasity令牌众包的声明

6.错误随机性

也被称为 没有什么是秘密的

合同对block.number年龄没有足够的验证，导致400个ETH输给一个未知的玩家，他在等待256个街区之前揭示了可预测的中奖号码。

阿森尼罗托夫

以太坊的随机性很难找到。虽然Solidity提供的功能和变量可以访问明显难以预测的值，但它们通常要么比看起来更公开，要么受到矿工影响。由于这些随机性的来源在一定程度上是可预测的，所以恶意用户通常可以复制它并依靠其不可预知性来攻击该功能。

损失：超过400 ETH

真实世界影响：

- SmartBillions彩票
- 运行

示例：

1. 甲智能合同使用块号作为随机有游戏用的源。
2. 攻击者创建一个恶意合约来检查当前的块号码是否是赢家。如果是这样，它就称为第一个智能合约以获胜; 由于该呼叫将是同一交易的一部分，因此两个合约中的块编号将保持不变。
3. 攻击者只需要调用她的恶意合约，直到获胜。

代码示例：

在第一个例子中，a private seed与iteration数字和keccak256散列函数结合使用来确定主叫方是否获胜。Eventhough的seed是private，它必须是通过交易在某个时间点设置，并因此在blockchain可见。

```
uint256 private seed;

function play() public payable {
        require(msg.value >= 1 ether);
        iteration++;
        uint randomNumber = uint(keccak256(seed + iteration));
        if (randomNumber % 2 == 0) {
                msg.sender.transfer(this.balance);
        }
}
```

在这第二个例子中，block.blockhash正被用来生成一个随机数。如果将该哈希值blockNumber设置为当前值block.number（出于显而易见的原因）并且因此设置为，则该哈希值未知0。在blockNumber过去设置为超过256个块的情况下，它将始终为零。最后，如果它被设置为一个以前的不太旧的区块号码，另一个智能合约可以访问相同的号码并将游戏合同作为同一交易的一部分进行调用。

```
function play() public payable {
        require(msg.value >= 1 ether);
```

```
        if (block.blockhash(blockNumber) % 2 == 0) {
            msg.sender.transfer(this.balance);
        }
}
```

其他资源：

- 在以太坊智能合约中预测随机数
- 在以太坊随机

7.前台运行

也被称为 检查时间与使用时间（TOCTOU），竞争条件，事务顺序依赖性（TOD）

事实证明，只需要150行左右的Python就可以获得一个正常运行的算法。

Ivan Bogatyy

由于矿工总是通过代表外部拥有地址（EOA）的代码获得gas费用，因此用户可以指定更高的费用以便更快地开展交易。由于以太坊区块链是公开的，每个人都可以看到其他人未决交易的内容。这意味着，如果某个用户正在揭示拼图或其他有价值的秘密的解决方案，恶意用户可以窃取解决方案并以较高的费用复制其交易，以抢占原始解决方案。如果智能合约的开发者不小心，这种情况会导致实际的和毁灭性的前端攻击。

真实世界影响：

- 班柯
- ERC-20
- 运行

示例：

1. 一个聪明的合同发布的RSA号（N = prime1 x prime2）。
2. 对其submitSolution()公共功能的调用与权利prime1并prime2奖励来电者。
3. Alice成功地将RSA编号考虑在内，并提交解决方案。
4. 有人在网络上看到爱丽丝的交易（包含解决方案）等待被开采，并以较高的gas价格提交。
5. 由于支付更高的费用，第二笔交易首先被矿工收回。该攻击者赢得奖金。

其他资源：

- 在以太坊智能合约中预测随机数
- 虚拟和解的前卫，悲痛和危险
- Frontrunning Bancor

8.时间篡改

也被称为 时间戳依赖

如果一位矿工持有合同的股份，他可以通过为他正在挖掘的矿区选择合适的时间戳来获得优势。

Nicola Atzei， Massimo Bartoletti和Tiziana Cimoli

从锁定令牌销售到在特定时间为游戏解锁资金，合同有时需要依赖当前时间。这通常通过Solidity中的block.timestamp别名或其别名完成now。但是，这个价值从哪里来？来自矿工！由于交易的矿工在报告采矿发生的时间方面具有回旋余地，所以良好的智能合约将避免强烈依赖所宣传的时间。请注意，block.timestamp有时（错误）也会在随机数的生成中使用，如＃6所述。错误的随机性。

真实世界影响：

- 政府

示例：

1. 一场比赛在今天午夜付出了第一名球员。
2. 恶意矿工包括他或她试图赢得比赛并将时间戳设置为午夜。
3. 在午夜之前，矿工最终挖掘该块。当前的实时时间“足够接近”到午夜（当前为该块设置的时间戳），网络上的其他节点决定接受该块。

代码示例：

以下功能只接受特定日期之后的呼叫。由于矿工可以影响他们区块的时间戳（在一定程度上），他们可以尝试挖掘一个包含他们交易的区块，并在未来设定一个区块时间戳。如果足够接近，它将在网络上被接受，交易将在任何其他玩家试图赢得比赛之前给予矿工以太：

```
function play() public {
        require(now > 1521763200 && neverPlayed == true);
        neverPlayed = false;
        msg.sender.transfer(1500 ether);
}
```

其他资源：

- 对以太坊智能合约的攻击调查
- 在以太坊智能合约中预测随机数
- 让智能合约变得更聪明

9短地址攻击

也被称为 涉及非连锁问题，客户端漏洞

为令牌传输准备数据的服务假定用户将输入20字节长的地址，但实际上并未检查地址的长度。

PawełBylica

短地址攻击是EVM本身接受不正确填充参数的副作用。攻击者可以通过使用专门制作的地址来利用这一点，使编码错误的客户端在将它们包含在事务中之前不正确地对参数进行编码。这是EVM问题还是客户问题？是否应该在智能合约中修复？尽管每个人都有不同的观点，但事实是，这个问题可能会直接影响很多以太网。虽然这个漏洞还没有被大规模利用，但它很好地证明了客户和以太坊区块链之间的交互带来的问题。其他脱链问题存在：重要的是以太坊生态系统对特定的javascript前端，浏览器插件和公共节点的深度信任。臭名昭着的链外利用被用于Coindash ICO的黑客在他们的网页上修改了公司的以太坊地址，诱骗参与者将ethers发送到攻击者的地址。

发现时间表：

日期	事件
2017年4月6日	如何通过阅读区块链来找到1000万美元

真实世界影响：

- 未知交换（s）

示例：

1. 交易所API具有交易功能，可以接收收件人地址和金额。
2. 然后，API transfer(address _to, uint256 _amount)使用填充参数与智能合约函数进行交互：它将12位零字节的地址（预期的20字节长度）预先设置为32字节长
3. 鲍勃（）要求爱丽丝转让他20个代币。他恶意地将她的地址截断以消除尾随的零。0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f00
4. Alice使用交换API和Bob（0x3bdde1e9fbaef2579dd63e2abbf0be445ab93f）的较短的19字节地址。
5. API用12个零字节填充地址，使其成为31个字节而不是32个字节。有效地窃取以下_amount参数中的一个字节。
6. 最终，执行智能合约代码的EVM将会注意到数据未被正确填充，并会在_amount参数末尾添加丢失的字节。有效地传输256倍以上的令牌。

其他资源：

- ERC20短地址攻击说明
- 分析ERC20短地址攻击
- 智能合同短地址攻击缓解失败
- 从标记中删除短地址攻击检查

10.未知的 未知物

我们相信更多的安全审计或更多的测试将没有什么区别。主要问题是评审人员不知道要寻找什么。

Christoph Jentzsch

以太坊仍处于起步阶段。用于开发智能合同的主要语言Solidity尚未达到稳定版本，而生态系统的工具仍处于试验阶段。一些最具破坏性的智能合约漏洞使每个人都感到惊讶，并且没有理由相信不会有另一个同样出乎意料或同样具有破坏性的漏洞。只要投资者决定将大量资金用于复杂而轻微审计的代码，我们将继续看到新的发现导致可怕的后果。对智能合约进行正式验证的方法尚不成熟，但它们似乎有望成为今日摇摇欲坠的现状。随着新类型的漏洞不断被发现，开发人员需要继续努力，并且需要开发新工具来在坏人之前找到它们。这个前10名可能会迅速发展，直到智能合约开发达到稳定和成熟的状态。

thanks & From:

http://www.dasp.co/

翻译：

慢雾安全团队

爱上平顶山

2018年05月18日