以太坊智能合约安全漏洞分析及对策

邱欣欣[1]，马兆丰[2]，徐明昆[1]
（1. 北京邮电大学网络技术研究院，北京市 100876
2. 北京邮电大学网络空间安全学院，北京市 100876）

[摘要] 智能合约是一种旨在以信息化方式传播、验证和执行合同的计算机协议。由于智能合约自带金融属性，执行过程中若出现漏洞会给用户及投资者带来较大困扰，因此如何编写安全可靠的智能合约至关重要。针对以太坊环境，对智能合约的漏洞进行了相关的研究和分析，提出了几种可能导致漏洞的常见编程陷阱，包括重入漏洞、整数溢出漏洞、拒绝服务漏洞、未检查 call 返回值以及短地址 / 参数攻击漏洞等。并有针对性地对这几种陷阱进行详细的原理分析和场景复现，并给出了相应的规避和解决的方法，提出了在安全模式下智能合约安全问题的解决方案。

[关键词] 区块链；以太坊；智能合约；安全漏洞；规避方法
[中图分类号] TP393;TP309　　[文献标识码] A　　[文章编号] 1009-0854（2019）02-0044-10

0 引言

2009 年，比特币推出并达到了巨额的资本化，引起了社会的广泛关注。在其发展过程中，人们渐渐意识到比特币的底层技术区块链的价值。区块链本质上是一个平等网络的分布式账本数据库。加密货币使用区块链作为公共账本，在链上记录所有的交易信息，这些交易经过系统大多数区块的认可，以取得整个网络的共识。大概 20 世纪 90 年代，尼克·萨博提出智能合约的概念。然而因为没有可信执行环境，这个技术始终没有被应用到实践产业中。而区块链技术的诞生为这个技术提供了使用基础。智能合约在区块链中有自身的账户，集合了代码和数据，可以实现一些特定的功能。抽象地说，智能合约是区块链上互不信任的用户之间达成共识的协议，它不依赖中间权威机构，而是通过区块链的共识机制来自动执行。

2015 年，以太坊产生，它首先看到了区块链和智能合约的契合并推出了白皮书《以太坊：下一代智能合约和去中心化应用平台》，成为智能合约最突出的框架。在以太坊中，智能合约以计算机程序的形式呈现，它可以自动执行，通常用图灵完备语言 solidity 编写，称为 EVM 字节码。智能合约在一定意义上是一组函数，每个函数由一系列的字节码指令定义，它可以对接收到的信息进行回应，可以接收和储存信息

和以太币，也可以向外发送信息和以太币。

为确保智能合约的正常执行，以太坊拥有共识协议，规定了区块链的增长机制。目前常用的三种共识算法为工作量证明机制、股权证明机制和授权股权证明机制。

智能合约的编写与传统的编程存在一定程度的不同，这极易造成一些潜在的漏洞。而且由于智能合约本身带有一些金融属性，可以存储大额的资产价值，这也造成它更易被恶意人员攻击并从中获利。智能合约存在漏洞的维护成本也相对传统程序更大，它部署在链上，一旦部署成功就不能修改，不能用传统的方法进行升级或者打补丁。这也意味着着对程序的编写人员要求更高，要在其设计和编码时就考虑到容错和异常的情况并进行处理。此外，由于以太坊为公有链，所有的信息公开，没有办法进行访问控制和中心化的审计，这也对智能合约的安全造成了更大的挑战。

在现在信息爆炸的社会，安全成为人们愈发关注的问题。本文主要总结并分析以太坊智能合约编写中容易发生的漏洞问题，以原理和实例的形式剖析具体的漏洞细节。

1 智能合约漏洞分析

1.1 重入漏洞

1.1.1 重入漏洞原理

调用和利用其他外部的合约是以太坊智能合约的重要特点之一。通常在合约中会发生一些将以太币发送给外部用户地址的行为，而转账和调用外部合约的操作需要智能合约的外部调用，这些外部调用若操作不慎，极其容易被攻击者利用，通过回退函数或者回调攻击合约自身来进行攻击和一些不当操作，造成用户的损失。比如之前引起以太币市场剧烈波动的 THE DAO 被攻击事件就是这种漏洞极为典型的例子。

类似于操作系统在进程调度中中断时出现的重入现象，可重入性通常表现为一个函数的同时多次调用，恶意合约在调用其他函数完成之前多次调用被攻击函数，在被攻击合约中“重新输入”了代码执行，从而实现攻击，这可能造成巨大的破坏。比如，当合约进行以太币转账时，攻击者在回调函数的外部地址处构建了一个恶意合约，当合约向这个外部地址进行转账时这个恶意的合约将被执行。此时控制权将从被攻击合约完全传递给恶意合约，此时恶意合约在拥有足够 Gas 的情况下再回调被攻击合约实现一次转账，同时再次被调用，如此循环将造成被攻击合约账户的巨大损失。

1.1.2 重入漏洞的代码复现

重入攻击通常发生于转账或调用外部合约，这段代码复现了重入攻击，展示了重入攻击具体的攻击方式。

```
contract Strorage{
  mapping(address => uint256) public balances;
  function deposit() public payable{
    balances[msg.sender] += msg.value;
  }
  function withdraw(uint256 withdrawAmount) public{
    require(balances[msg.sender] >= withdrawAmount);
    require(this.balance > withdrawAmount);
    require(msg.sender.call.value(withdrawAmount)());
    balances[msg.sender] -= withdrawAmount;
  }
}
```

这段代码简单实现了存款和取款的操作。用户可以进行存钱，合约会记录每个用户的存款额度，同时用户可以通过 withdraw 函数取出自己的存款。这段代码似乎在逻辑上没有漏洞，但是问题出现在进行存款回收转账时的 call.value() 函数上。以太坊智能合约漏洞与自身语法特性有很大关系，与具有同样转账功能的 send() 和 transfer() 函数只有 2300gas 处理转账操作不同，call.value() 函数会将合约剩余的所有 gas 全部用于外部调用。若转币时目标地址是一个合约，那么会自动调用合约中的 fallback 函数，若攻击者在部署一个恶意递归调用转账操作的合约，则会将公共钱包合约里的余额全部提出来，对被攻击合约造成巨大损失。

```
contract Attack{
    Strorage public etherStorage;
    function attackStorage() public payable{
        require(msg.value >= 1 ether);
        etherStorage.deposit.value(1 ether)();
        etherStorage.withdraw(1 ether);
    }
    function () payable {
        if(etherStorage.balance > 1 ether){
            etherStorage.withdraw(1 ether);
        }
    }
}
```

具体攻击过程为恶意合约向公共钱包合约存入一个以太币，然后调用 withdraw 函数进行回收，在公共钱包合约向恶意合约成功转账一个以太币之后，自动调用恶意合约的 fallback 函数，此时在回调函数中又一次调用 withdraw 函数，因为此时在公共钱包合约中恶意合约账户的余额未被清零，所以依然会调用成功，如此循环调用，恶意合约会将公共钱包合约中的以太币全部转出来完成攻击。

1.1.3 重入漏洞规避和解决的方法

为了避免重入漏洞，我们在执行转账操作时要仔细权衡 send()、transfer() 和 call.value() 函数的使用。call.value() 函数会给执行账户所有可用的 gas，通过这种方式来进行转账存在可重入的风险。而 send() 和 transfer() 函数给执行账户仅有 2300gas，这只够捕获一个 event，这可以有效保证可重入安全，因此在进行转账时建议优先采用这种方式。这种方式存在的一个问题是可能导致合约调用 fallback 函数时因 gas 不足出现问题，一种有效的平衡方式是 pull 和 push 机制。在账户对外转账，即 push 时采用 send() 或者 transfer() 函数，在合约内转账操作，即 pull 时采用 call.value() 函数。通常将外部调用隔离到合约内部的交易中可以有效避免外部调用失败引起的损失，调用发起的合约只负责初始化外部调用，具体操作由对方完成。比如在转账时事先设置对方可以从当前账户撤回的数量，然后让对方自己撤回资产而不是主动转账给他，这也可以避免产生 gas 不足的问题。

在智能合约的编写中最好尽量避免外部合约的调用，这极易被攻击者利用。外部合约可能会在被攻击合约或者合约内部执行恶意代码，引发一系列的错误和损失，所以尽可能在智能合约中移除外部调用。如果因业务或功能需要而不得不调用时，尽量检查调用的返回值来对可能出现的错误进行处理，同时标记不受信任

的合约。不要假设知道外部调用的控制流程，因为不受信任的合约可能存在恶意代码，或者它调用的其他合约存在恶意代码，这可能挟持控制流程导致一系列的错误。同时正确使用 require() 和 assert() 函数来进行条件限制和报错返回，有效规避和处理潜在的错误。

1.2 整数溢出漏洞

1.2.1 整数溢出漏洞原理

在合约中一个整型变量只能有一定范围的数字表示区间。比如 uint8 只能存储范围在 0 到 255 的数字，若尝试存储 256 则会上溢导致错误存储为 0，同样 (uint8)0–1 将导致下溢被错误存储为 (uint8)255。同样的，有符号整数比如 int8 类型，表示范围为 –128 到 127，当想存储 128 时，进位符号位变为 1，此时会被错误存储为 –128，当想存储 –129 时，(int8)(–128)–1 会出现下溢错误存储为 127。

这是因为以太坊虚拟机为整数指定了固定的大小，当在操作数据时超出了数据范围会出现溢出错误或 gas 不足，这会使得攻击者有机会滥用代码并创建不合理的逻辑流程。

1.2.2 整数溢出场景

在智能合约中对变量值进行操作时若不注意数值范围极易发生溢出现象。

```
contract arithmetic_issues{
    mapping(address => uint) balances;

    function deposit() payable {
        balances[msg.sender] += msg.value;
    }

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

在 withdraw 函数中通过 require(balances[msg.sender] – amountToWithdraw > 0) 这一句来验证账户的余额是否足够，这句代码里存在明显的整数溢出漏洞。在智能合约中 uint 为 256 位无符号整数，数据表示范围为 0 到 2**256–1，若传入的 amountToWithdraw 参数数值大于账户中的余额则会出现整数下溢，被转出大于用户余额的比特币。

绕过算术操作来进行逻辑判断，可以将代码中的判断条件改为 require(balances[msg.sender] > amountToWithdraw)，这可以相应地避免代码造成的整数溢出，规避漏洞的发生。

1.2.3 整数溢出漏洞规避和解决的方法

为了防止整数溢出，通常可以在算数逻辑前后进行验证，或者可以使用或建立可以取代标准数学运算符的数学库，比如 openzeppelin 的智能合约函数库中的 SafeMath。

智能合约中除了算术操作要规避溢出错误之外，还要注意数组以及循环变量的大小，其中数组为 256 位无符号整型，循环变量为 8 位无符号整型，这些变量若操作不慎也极易发生溢出漏洞。

1.3 拒绝服务漏洞

1.3.1 拒绝服务漏洞原理

拒绝服务攻击即 DOS，通常是使得原来的

代码逻辑无法运行，导致以太币和 gas 的消耗。这种攻击手段的方式有很多，通常导致的结果都是使得合约无法正常运行，无法提供正常服务。比如在回调函数中触发报错操作，使得合约无法继续运行下去，或者恶意更改变量造成 gas 的枯竭中止合约服务等。

1.3.2 拒绝服务漏洞场景

一些细微的智能合约编码方式或逻辑都极有可能被攻击者利用造成 DOS 攻击。比如触发报错操作，中止合约服务。

```
contract becomeowner{
    address public owner;
    uint money;

    function beowner() payable{
        require(msg.value >= money);
        owner.transfer(money);
        owner = msg.sender;
    }
}
```

这个合约实现了用户通过资产竞争 owner 的逻辑。若是攻击者在成功成为 owner 之后在其 fallback 函数中触发会产生错误的操作，则会中止合约的正常运行，一直处于 owner 的状态，造成 DOS 攻击。

另一种容易造成 DOS 漏洞的情况是将代码控制权集中在某一个变量上，这极易造成某一个用户的控制权垄断，当其不活动或者出现丢失密钥的情况下，会造成整个智能合约的瘫痪无法继续运行，下面这段代码给出了一个基本的攻击演示。

```
bool public isSubmit = false;
address public user;

function submit() public{
    require(msg.sender == user);
    isSubmit = true;
}

function transfer(address _to, uint _value) returns (bool){
    require(isSubmit);
    super.transfer(_to,_value);
}
```

代码中若用户不在活动，或丢失密钥，则 isSubmit 控制权将一直在该用户处，user 无法调用 transfer 函数转账，整个智能合约停滞。

还有一种 DOS 漏洞出现在数组循环中，恶意合约容易通过外部操纵映射或者数组来干预智能合约的正常运行，使得 gas 枯竭，造成合约无法继续运行下去。

除此之外，智能合约在进行外部调用时也极易发生 DOS 攻击。比如在合约向外部合约发送以太币时对方不接受或者在请求外部输入时没有正常输入，则会导致智能合约永远无法转换为新的状态而出错。

1.3.3 拒绝服务漏洞规避和解决的方法

在第一个和第二个出现漏洞的情况中，如果要进行预防和规避，可以引入时间的机制，允许用户在一定时间内完成操作。要考虑到用户操作失败的情况，若超时或操作失败等采用一定的策略来让合约继续运行下去，而不是被锁在失败的地方。在外部调用出现问题时也可以采用这种机制。

而在易被恶意合约操纵外部映射或者数组

的情况下，最好采用 pull 而不是 push 机制，让每个用户自己去 pull 自己的资产，而不是通过循环一个极易被外部操纵的数据结构去一个一个 push。

DoS 在 solidity 中无处不在，通常表现为不可恢复的恶意操作或者可控制的无限资源消耗，导致 Ether 和 Gas 的大量消耗，让原本的合约代码逻辑无法正常运行。因此采取一定的预防机制来避免这种漏洞是必要的。

1.4 未检查 call 返回值

1.4.1 未检查 call 返回值漏洞原理

在进行外部调用时，若未对返回值进行检查，极易产生漏洞。

底层的调用有三种，第一种是 call()，它会返回一个 bool 值来表明是否调用成功，通常用 <addr>.call(bytes4(keccak(“func(params)”)))。

第二种是 delegatecall()，和 call() 函数一样，它也会返回一个 bool 值，不同的是它会将外部代码直接作用于合约的上下文。

第三种是 callcode() 函数，与 delegatecall 一样，会将外部代码作用于上下文，但是它在 msg.sender() 和 msg.value 上存在一点差异。

1.4.2 未检查 call 返回值规避和解决的方法

在智能合约的编写中，最好将外部的发送功能与智能合约内部的其他功能分离开来，采取 pull 模式，让用户自己去调用撤销的操作，而不是合约主动去向外部发送。这样可以让用户独立处理失败后的后果，从而不会导致智能合约的错误运行。调用 pull 功能的最终用户会负担失败的可能操作。

除此之外，尽可能在转账操作中使用 transfer() 函数而不是 send()。同时在每次调用之后都要检查返回值以便及时处理可能存在的错误。

1.5 短地址 / 参数攻击

1.5.1 短地址 / 参数攻击漏洞原理

2017 年 4 月，Golem 项目在一篇博文中提及发现了一个影响交易的安全漏洞。根据该帖子，当某些交易所处理 ERC20 令牌的交易时，没有对账户地址长度进行输入验证。这导致传送给智能合约转账函数的参数异常，以及传输金额的一系列错误问题。攻击者极易盗取 token 的交易账户。

在交易所提币或者钱包转账等场景下极易发生短地址攻击，这种攻击行为极易发生在接受畸形地址的地方。它并不是专门针对智能合约，而是针对与之交互的第三方应用程序。在编写智能合约的时候需要严格验证输入数据的正确性，在非链上的功能上也要对用户所输入的地址格式进行验证，防止短地址攻击。

将参数传递给智能合约时，EVM 会对参数进行处理，根据 ABI 规范对其进行编码。用户可以发送比预期参数长度短的编码参数，在这种情况下，以太坊虚拟机会将 0 填到编码参数的末尾以弥补预期的长度。当第三方应用程序不验证输入时，这会成为问题。

1.5.2 短地址 / 参数攻击漏洞场景

Golem 项目解释的攻击举例说明了一个相当独特的案例，其中交易既充当客户端又充当服务器。也就是说，交易既是用户购买 token 的服务器，也是以太坊网络的客户端。这与传统的客户端直接使用以太坊网络的合约交易不同，任何交易错误都来自于客户端而不是第三方。幸运的是，对

于 Golem 项目来说，这个漏洞并不为人所知。此后它被称为“ERC20 短地址攻击”。

短地址 / 参数攻击极易发生在合约转账的场景下。

```
contract MyToken {
    mapping (address => uint) balances;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    function MyToken() {
        balances[tx.origin] = 10000;
    }

    function sendCoin(address to, uint amount) returns(bool sufficient) {
        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[to] += amount;
        Transfer(msg.sender, to, amount);
        return true;
    }

    function getBalance(address addr) constant returns(uint) {
        return balances[addr];
    }
}
```

调用转账函数后，msg.data 为

0x90b98a11 是方法 ID（4 个字节），它是方法签名的 Keccak（SHA-3）哈希。0000000000000000000000062bec9abe373123b9b635b75608f94eb8644163e 是被转账的地址，32 个字节。

0000000000000000000000000000000000000000000000000000000000000002 是转账金额，32 个字节。

在这种情况下，如果客户发送错误的信息，例如不正确填充地址使得地址小于 32 个字节，总输入数据长度将小于预期，这将导致可能发生数据下溢的情况。例如，再次发送一些硬币到 0x62bec9abe373123b9b635b75608f94eb8644163e 中时，丢弃地址中的最后一个字节“3e”，将会得到以下输入数据：

0x90b98a11

0000000000000000000000062bec9abe373123b9b635b75608f94eb86441600

0000000000000000000000000000000000000000000000000000000000000002，最后缺失了 1 个字节。当数据进入 EVM 中进行处理时，会将参数最后补 0，将参数进行向左 8 位的位移操作，丢失的字节会被 00 替换。结果，地址在末尾添加了 00，需要转账的金额变为了 512。

1.5.3 短地址 / 参数攻击规避和解决的方法

目前对此漏洞的认识使得它被利用的可能性大大降低，但是如果正在考虑将代币供应交给可能不信任的交易所，则在 ERC20 合同中执行输入验证是必要的。这可以极大可能地防止这些类型的攻击。此外，参数排序在这个漏洞的预防中起着重要的作用。由于填充只发生在最后，则智能合约中参数的合理排序将在一定程度上预防短地址攻击。

2 总结

2.1 常用的保证智能合约安全的工具和方法

现在有许多开源的安全智能合约框架，

Zeppelin 是其中比较出色的一个。它侧重于 solidity 语言，与以太坊首要构建架构 Truffle 相集成，提供了拉动式支付模块、自动检测漏洞、可重用基础组件等安全模块，大大提升了智能合约的编写效率和安全性。

除此之外，开发者还可以用智能合约安全分析工具 Oyente 来对自己的代码进行检测，用来检查合约中特定的安全属性或者存在的一些 bug。

由于智能合约安全与传统编程的特殊性，它存在一些不是 bug 但是有安全隐患的场景，比如外部调用，而 Oyente 也可以对这种场景进行检测和分析，这大大提升了智能合约的安全性，利于发现容易被忽视的一些代码场景，规避可能存在的风险和漏洞。

另一种提升智能合约安全性的技术就是形式化验证，采用数学证明来验证源代码实现了某种形式的规范或达到了某种要求。

通常对智能合约的形式化验证都要先对智能合约的功能进行形式化描述，然后对代码进行形式化描述，最后进行证明，以此来验证智能合约的可行性和安全性，在一定程度上来保护智能合约的安全。

2.2 智能合约安全开发的建议

智能合约漏洞通常分为三类。

第一类是 Solidity 语言层面的漏洞，比如重入漏洞、未检查 call 返回值、类型混乱、拒绝服务、访问控制、整数的溢出等漏洞。

第二类是以太坊虚拟机层面的漏洞，比如交易过程中以太币丢失、堆栈调用深度限制、合约运行中 gas 的成本限制以及变量的使用类型限制等漏洞。

第三类是区块链本身的漏洞，比如交易顺序依赖、时间戳依赖以及随机数的生成等漏洞。

具体分类如表 1。

表 1 智能合约漏洞分类

漏洞分类	漏洞发生原因
Solidity 层面	重入漏洞
	未检查 call 返回值
	整数溢出
	类型混乱
	拒绝服务
	访问控制
EVM	变量类型限制
	合约运行 Gas 成本
	交易过程中以太币丢失
	调用堆栈深度
区块链	时间戳依赖
	交易顺序出错
	随机数生成错误

在第二章中已经对一些漏洞情况进行了深入的分析。在具体的智能合约编写中还是要谨慎设计并编码，避免因人工失误而造成不可挽回的损失。必要的情况下，可以尽量简化智能合约的复杂性，用简单的合约来代替复杂的合约。同时在智能合约功能实现之后，采用尽可能丰富和细致的测试方式对合约进行测试，对业务流程、逻辑等方面要进行详尽的测试流程和安全性检测，以保证智能合约的安全性。

参考文献：

[1] Frantz, C.K., Nowostawski, M. “From institutions to code: towards automatedgeneration of smart contracts.” Workshop on Engineering

Collective AdaptiveSystems (eCAS). 2016.

[2] Namoto S. "Bitcoin : A peer-to-peer electronic cash system." http://bitcoin.org/bitcoin.pdf , 2008.

[3] Ethereum Classic. https://ethereumclassic.github.io/.

[4] Ethereum reddit page. https://www.reddit.com/r/ethereum.

[5] MAker DART. "a random number generating game for Ethereum." https://github.com/makerdao/maker-darts/.

[6] Marino, B., Juels, A. "Setting standards for altering and undoing smart contracts." RuleML, 2016:151 - 166.

[7] Buterin, V. "Ethereum: a next generation smart contract and decentralized application platform." https://github.com/ethereum/wiki/wiki/White-Paper.2013.

[8] Wood, G. "Ethereum: a secure decentralised generalised transaction ledger." http://gavwood.com/paper.pdf. 2014.

[9] RANDAO, "a DAO working as RNG of Ethereum." https://github.com/randao/randao/.

[10] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A. "Making smart contractssmarter." ACM CCS. http://eprint.iacr.org/2016/633/. 2016.

[11] Etherscripter. http://etherscripter.com.

[12] C.Boyd, C.Carr. "Fair client puzzles from the bitcoin blockchain. Australasian Conference on Information Security and privacy." Switzcrland : Springer, 2016:161-177.

[13] Luu, L., Teutsch, J., Kulkarni, R., Saxena, P. "Demystifying incentives in the consensus computer." ACM CCS, 2015:706 - 719.

[14] Hirai, Y. "Formal verification of Deed contract in Ethereum name service." https://yoichihirai.com/deed.pdf.

[15] Gervais, A., Karame, G.O., W ¨ ust, K., Glykantzis, V., Ritzdorf, H., Capkun, S. "On the security and performance of proof of work blockchains." ACM CCS, 2016. 3 - 16.

[16] Eyal, I., Sirer, E. "Majority is not enough: bitcoin mining is vulnerable." FinancialCryptography and Data Security, 2014, 436 - 454.

[17] Ehtereum. http://www.ethereum.org/. 2017.

[18] Anderson, L., Holz, R., Ponomarev, A., Rimba, P., Weber, I. "New kids on theblock: an analysis of modern blockchains." CoRR, abs/1606.06530.2016.

[19] W ¨ ust, K., Gervais, A. "Ethereum Eclipse Attacks." Technical report, ETH-Z ¨ urich. 2016.

[20] Karl. "Security of Blockchain Technologies." Ph.D. thesis, Swiss Federal Institute of Technology. 2016.

作者简介

邱欣欣，北京邮电大学网络技术研究院硕士，主要研究方向为区块链及安全技术研究与开发。

马兆丰，北京邮电大学区块链及安全技术联合实验室主任，研究方向是区块链及安全技术、移动互联网与大数据安全创新、网络与信

息安全、数字版权管理。

徐明昆，北京邮电大学网络技术研究院信息网络中心研究生导师，高级工程师，主要研究方向为云计算与中间件。

Ethereum Smart Contract Security Vulnerability Scenario Analysis

QIU Xinxin[1], Ma Zhaofeng[2], Xu Mingkun[1]

(1.Institute of network technology, Beijing University of Posts and Telecommunications, BeiJing ,100876;
2. School of Cyberspace Security, Beijing University of Posts and Telecommunications, BeiJing,100876)

[Abstract] Intelligent contract is a computer protocol that aims to propagate, verify and execute contracts through information technology. Since intelligent contracts have their own financial attributes, if there are loopholes in the execution process, users and investors will be greatly troubled, so how to write safe and reliable intelligent contracts is very important.Aiming at the environment of ETF, the vulnerabilities of intelligent contract are studied and analyzed. Several common programming traps which may lead to vulnerabilities are proposed, including re-entry vul-nerabilities, integer overflow vulnerabilities, denial of service vulnerabilities, unchecked call return values and short address/parameter attack vulnerabilities.In addition, the detailed principle analysis and scene reproduction of these traps are carried out, and the corresponding methods of avoiding and resolving them are given. A solution to the security problem of intelligent contract under the security mode is put forward.

[Keywords] BlockChain; Ethereum; Smart Contracts; Security Vulnerability; Circumvention Method