Solidity智能合约开发知识

第8.2课：智能合约安全基础

学习目标：掌握智能合约中最常见的安全漏洞、理解攻击原理和防御方法、学会使用安全检查清单、能够在实际开发中避免安全陷阱

预计学习时间：3-4小时

难度等级：中级

重要提示：这些都是真实存在的漏洞，已造成数十亿美元损失！安全是智能合约的生命线，一个漏洞可能导致数百万美元损失。

目录

---

1. 智能合约安全概述

1.1 为什么安全如此重要

智能合约一旦部署到区块链上，代码就无法修改。这意味着：

1. **不可逆性**：
   - 一旦发现漏洞，无法直接修复
   - 只能通过升级机制（如代理模式）或硬分叉来解决
   - 损失往往是永久性的
2. **资金规模巨大**：
   - DeFi协议管理着数十亿甚至数百亿美元的资金
   - 一个漏洞可能导致整个协议的资金被掏空
   - 影响范围可能波及整个生态系统
3. **公开透明**：
   - 所有代码都在链上公开
   - 攻击者可以仔细研究代码寻找漏洞
   - 没有"隐藏"的安全措施
4. **自动化执行**：
   - 智能合约自动执行，无需人工干预

- 攻击一旦成功，资金会立即被转移
- 没有"后悔"的机会

历史教训：

- **The DAO攻击（2016）**：损失约6000万美元，导致以太坊硬分叉
- **Parity钱包漏洞（2017）**：两次攻击，共损失约1.8亿美元
- **多个代币溢出漏洞（2018）**：导致代币增发，市值蒸发
- **多个DeFi协议被攻击**：损失金额从数百万到数亿美元不等

1.2 常见安全漏洞类型

本课程将重点讲解以下五种最常见的安全漏洞：

1. **重入攻击（Re-entrancy）**：
   - 最危险的漏洞之一
   - 利用外部调用和状态更新的顺序问题
   - The DAO攻击就是典型案例
2. **整数溢出（Integer Overflow）**：
   - 数值运算超出数据类型范围
   - Solidity 0.8.0后已内置保护
   - 但仍需注意unchecked块的使用
3. **权限控制漏洞（Access Control）**：
   - 缺少权限检查
   - 任何人都可以调用关键函数
   - Parity钱包漏洞就是典型案例
4. **拒绝服务攻击（DoS）**：
   - Gas耗尽导致功能不可用
   - 状态阻塞攻击
   - 循环处理大量数据
5. **前端运行攻击（Front-Running）**：
   - 利用交易顺序依赖
   - MEV（最大可提取价值）攻击
   - 交易被抢跑

1.3 安全开发的基本原则

在深入学习具体漏洞之前，我们先了解安全开发的基本原则：

1. **最小权限原则**：
   - 只给必要的权限
   - 关键函数必须有权限检查
   - 使用OpenZeppelin的Ownable或AccessControl
2. **防御性编程**：
   - 假设外部调用是恶意的
   - 假设输入数据是恶意的
   - 永远验证所有输入

3. **检查-效果-交互模式（CEI）**：
    - 先检查所有条件
    - 再更新状态
    - 最后进行外部调用
4. **使用经过审计的库**：
    - OpenZeppelin是行业标准
    - 不要重复造轮子
    - 使用经过充分测试的代码
5. **代码审计**：
    - 专业安全公司审计
    - 社区代码审查
    - 自动化安全工具扫描

---

2. 重入攻击（Re-entrancy）

重入攻击是智能合约中最常见和最危险的安全漏洞之一。2016年的The DAO攻击就是利用了这个漏洞，导致损失了价值6000万美元的以太币。

2.1 什么是重入攻击

重入攻击的定义：

重入攻击（Re-entrancy Attack）是指恶意合约在外部调用返回之前，再次调用原合约的函数，利用状态未更新的漏洞进行攻击。

为什么叫"重入"：

函数还没执行完，就被"重新进入"了！攻击者利用外部调用触发的回调函数，在状态更新之前再次调用原函数。

攻击原理：

```
正常流程：
用户 → withdraw() → 更新余额 → 转账 → 完成

重入攻击流程：
攻击者 → withdraw()
    ↓
  检查余额：10 ETH（通过）
    ↓
  转账 10 ETH → 触发receive()
                    ↓
                再次调用withdraw()
                    ↓
                  检查余额：10 ETH（还没更新！）
                    ↓
                  再转账 10 ETH...
                    ↓
                  继续重入...
                    ↓
```

```
                    直到余额耗尽
```

2.2 不安全的代码示例

让我们看一个存在重入漏洞的银行合约：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 存在重入漏洞的银行合约
contract VulnerableBank {
    // 记录每个用户的余额
    mapping(address => uint256) public balances;

    // 事件：记录存款和提现操作
    event Deposit(address indexed user, uint256 amount);
    event Withdrawal(address indexed user, uint256 amount);

    /**
     * @notice 存款函数
     * @dev 用户可以向合约存入以太币
     */
    function deposit() external payable {
        // 更新用户余额
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    /**
     * @notice 提现函数（存在重入漏洞！）
     * @dev 危险：先转账，后更新状态
     */
    function withdraw() external {
        // 步骤1：检查用户余额
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        // 步骤2：向用户转账（外部调用）
        // 危险：在状态更新之前进行外部调用
        // 如果msg.sender是恶意合约，它可以在receive函数中再次调用withdraw
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        // 步骤3：更新余额（太晚了！）
        // 危险：状态更新在外部调用之后
        // 如果发生重入，此时余额还没有被清零，攻击者可以再次提取
        balances[msg.sender] = 0;

        emit Withdrawal(msg.sender, amount);
    }
```

```
    /**
     * @notice 查询合约总余额
     */
    function getContractBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

问题分析：

1. **外部调用在状态更新之前**：
   - `call`会触发接收者的`receive`或`fallback`函数
   - 如果接收者是恶意合约，可以在回调中再次调用`withdraw`
   - 此时`balances[msg.sender]`还没有被清零
2. **状态检查失效**：
   - 重入时，余额检查仍然通过
   - 因为余额还没有被更新
   - 攻击者可以重复提取资金

2.3 攻击合约示例

以下是一个利用重入漏洞的攻击合约：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 攻击合约：利用重入漏洞
contract Attacker {
    // 存储目标银行合约地址
    VulnerableBank public vulnerableBank;

    // 记录攻击次数，避免Gas耗尽
    uint256 public attackCount;

    // 构造函数：初始化目标合约地址
    constructor(address _vulnerableBank) {
        vulnerableBank = VulnerableBank(_vulnerableBank);
    }

    /**
     * @notice 接收以太币时触发重入攻击
     * @dev 这是攻击的关键：在receive函数中再次调用withdraw
     */
    receive() external payable {
        // 限制攻击次数，避免Gas耗尽
        // 如果攻击次数少于3次，且银行还有余额，继续攻击
        if (attackCount < 3 && address(vulnerableBank).balance > 0) {
            attackCount++;
            // 重入攻击：再次调用withdraw
            // 此时vulnerableBank的balances[address(this)]还没有被清零
```

```
            vulnerableBank.withdraw();
        }
    }

    /**
     * @notice 发起攻击
     * @dev 攻击流程：先存款，再提现，在receive中触发重入
     */
    function attack() external payable {
        require(msg.value >= 1 ether, "Need at least 1 ether");
        attackCount = 0;

        // 步骤1：先向银行存入1 ether
        // 这样攻击者就有了1 ether的余额
        vulnerableBank.deposit{value: msg.value}();

        // 步骤2：发起第一次提现
        // 这会触发receive函数，在receive中会再次调用withdraw
        vulnerableBank.withdraw();
        // 在receive函数中会触发多次重入调用
    }

    /**
     * @notice 查询攻击者余额
     */
    function getBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

攻击流程详解：

```
1. 攻击者调用attack()，存入1 ether
   - vulnerableBank.balances[attacker] = 1 ether

2. 攻击者调用withdraw()
   - 检查余额：1 ether（通过）
   - 向攻击者转账1 ether
   - 触发攻击者的receive()函数

3. receive()函数中再次调用withdraw()
   - 此时balances[attacker]还是1 ether（还没被清零！）
   - 检查余额：1 ether（通过）
   - 再次向攻击者转账1 ether
   - 再次触发receive()函数

4. 重复步骤3，直到攻击次数达到限制
   - 最终攻击者提取了4 ether（1 ether本金 + 3 ether窃取）
```

2.4 The DAO攻击案例分析

事件概况：

- **时间**：2016年6月17日
- **损失金额**：约 $60,000,000 美元
- **被盗ETH**：约 3,600,000 ETH（总量的15%）
- **攻击时间**：持续数小时
- **影响范围**：整个以太坊生态系统

漏洞代码分析：

The DAO的`splitDAO`函数存在重入漏洞：

```
// The DAO的splitDAO函数简化版
function splitDAO() {
    uint256 amount = balanceOf[msg.sender];

    // 漏洞：先转账
    if (msg.sender.call.value(amount)()) {
        // 漏洞：后更新余额
        balanceOf[msg.sender] = 0;
    }
}
```

攻击者的恶意合约：

```
contract Attacker {
    function() payable {
        // 在receive函数中再次调用splitDAO
        if (dao.balance > 0) {
            dao.splitDAO();  // 重入!
        }
    }
}
```

事件影响和后果：

1. **时间线**：
    - 6月17日：攻击开始 → 社区发现 → 紧急应对
    - 6月18日-7月：讨论解决方案 → 社区分歧 → 投票决定
    - 7月20日：硬分叉执行 → 资金返还
2. **结果**：
    - **ETH（以太坊）**：硬分叉后，资金被返还
    - **ETC（以太坊经典）**：保留原链，攻击者保留了资金
3. **长期影响**：
    - 智能合约安全成为重点
    - 代码审计变得必不可少
    - 重入防护成为标准实践
    - 社区分裂（ETH vs ETC）

三大教训：

1. **代码审计必不可少**：
   - The DAO代码经过审查，但仍有漏洞
   - 需要专业安全公司审计
2. **不可变性的双刃剑**：
   - 合约无法修改，安全必须第一次就做对
   - 硬分叉是最后手段
3. **防御性编程**：
   - 假设外部调用是恶意的
   - 先更新状态，后外部调用
   - 使用经过验证的模式

2.5 防御方法

方法1：CEI模式（Checks-Effects-Interactions）

CEI模式是防止重入攻击的金科玉律。它要求按照以下顺序组织代码：

1. **Checks（检查）**：验证所有条件
2. **Effects（效果）**：更新所有状态
3. **Interactions（交互）**：最后才外部调用

安全的代码示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 安全的银行合约：遵循CEI模式
contract SecureBank {
    // 记录每个用户的余额
    mapping(address => uint256) public balances;

    // 事件：记录存款和提现操作
    event Deposit(address indexed user, uint256 amount);
    event Withdrawal(address indexed user, uint256 amount);

    /**
     * @notice 存款函数
     */
    function deposit() external payable {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    /**
     * @notice 安全的提现函数：遵循CEI模式
     * @dev 关键：先更新状态，后外部调用
     */
    function withdraw() external {
```

```
        // 1. Checks（检查）：验证所有条件
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        // 2. Effects（效果）：先更新状态
        // 关键：在外部调用之前更新状态
        // 这样即使发生重入，余额检查也会失败
        balances[msg.sender] = 0;

        // 3. Interactions（交互）：然后进行外部调用
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        emit Withdrawal(msg.sender, amount);
    }

    /**
     * @notice 查询合约总余额
     */
    function getContractBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

为什么CEI模式安全：

- 余额在外部调用前就是0
- 即使发生重入，余额检查也会失败（`require(amount > 0)`会失败）
- 攻击无效

方法2：重入锁（ReentrancyGuard）

重入锁使用互斥锁防止重入调用。

自定义重入锁：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 使用重入锁的银行合约
contract SecureBankWithLock {
    mapping(address => uint256) public balances;

    // 重入锁：使用布尔变量作为锁
    bool private locked;

    event Deposit(address indexed user, uint256 amount);
    event Withdrawal(address indexed user, uint256 amount);

    /**
     * @notice 重入锁修饰符
     * @dev 防止函数被重入调用
```

```
     */
    modifier noReentrant() {
        // 检查锁是否已锁定
        require(!locked, "No re-entrancy");
        // 设置锁
        locked = true;
        // 执行函数
        _;
        // 释放锁
        locked = false;
    }

    /**
     * @notice 存款函数
     */
    function deposit() external payable {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    /**
     * @notice 安全的提现函数：使用重入锁
     * @dev 即使不遵循CEI模式，重入锁也能提供保护
     */
    function withdraw() external noReentrant {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        balances[msg.sender] = 0;

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        emit Withdrawal(msg.sender, amount);
    }
}
```

重入锁的工作原理：

1. 第一次调用时，`locked = false`，通过检查
2. 设置`locked = true`
3. 如果发生重入，`locked`已经是`true`，检查失败
4. 函数执行完成后，释放锁

方法3：使用OpenZeppelin的ReentrancyGuard

OpenZeppelin提供了经过审计的重入锁实现，推荐使用：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 导入OpenZeppelin的ReentrancyGuard
```

```solidity
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

// 使用OpenZeppelin重入锁的银行合约
contract SecureBankWithOpenZeppelin is ReentrancyGuard {
    mapping(address => uint256) public balances;

    event Deposit(address indexed user, uint256 amount);
    event Withdrawal(address indexed user, uint256 amount);

    /**
     * @notice 存款函数
     */
    function deposit() external payable {
        balances[msg.sender] += msg.value;
        emit Deposit(msg.sender, msg.value);
    }

    /**
     * @notice 安全的提现函数：使用OpenZeppelin的nonReentrant修饰符
     * @dev OpenZeppelin的实现经过充分审计，更安全可靠
     */
    function withdraw() external nonReentrant {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        balances[msg.sender] = 0;

        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        emit Withdrawal(msg.sender, amount);
    }
}
```

OpenZeppelin ReentrancyGuard的优势：

- 经过充分审计
- 被广泛使用
- Gas优化良好
- 开箱即用

方法对比：

方法	安全性	Gas成本	易用性
CEI模式	高	最低	需理解
自定义锁	高	较低	容易
OpenZeppelin	高	低	最简单

最佳实践：

1. 优先使用CEI模式（零成本）
2. 关键函数添加ReentrancyGuard
3. 使用OpenZeppelin库（不要自己写）
4. 代码审计必不可少

---

3. 整数溢出（Integer Overflow）

整数溢出是另一个常见的安全漏洞。虽然Solidity 0.8.0后已经内置了溢出保护，但理解溢出原理和正确使用unchecked块仍然非常重要。

3.1 什么是整数溢出

整数溢出的定义：

整数溢出（Integer Overflow/Underflow）是指数值运算的结果超出了数据类型的范围，导致结果"回绕"到范围的另一端。

可视化示例：

```
uint8 类型范围：0 到 255（8位，2^8）

 0 ──────────────────────────────→ 255
 ↑                                 ↓
 └─────────────────←─────────────────┘

上溢（Overflow）：
255 + 1 = 0  （回绕到起点）

下溢（Underflow）：
0 - 1 = 255  （回绕到终点）
```

溢出类型：

1. **上溢（Overflow）**：
   - 数值超过最大值
   - 例如：`uint8`的最大值是255，`255 + 1 = 0`
2. **下溢（Underflow）**：
   - 数值小于最小值
   - 例如：`uint8`的最小值是0，`0 - 1 = 255`

3.2 不安全的代码示例

在Solidity 0.7.6及以前的版本中，溢出不会自动检查，需要手动使用SafeMath库。

存在溢出漏洞的代币合约：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;
```

```
//  Solidity 0.7.6 代币合约漏洞示例
contract UnsafeToken {
    // 记录每个地址的代币余额
    mapping(address => uint256) public balanceOf;

    /**
     * @notice 转账函数（存在溢出漏洞！）
     * @dev 危险：没有溢出检查
     */
    function transfer(address to, uint256 amount) public {
        //  危险：可能发生下溢
        // 如果msg.sender的余额小于amount，会发生下溢
        // 例如：balanceOf[msg.sender] = 0, amount = 1
        // 结果：0 - 1 = 2^256 - 1（天文数字！）
        balanceOf[msg.sender] -= amount;

        //  危险：可能发生上溢
        // 如果to的余额加上amount超过uint256的最大值，会发生上溢
        // 结果会回绕到很小的数值
        balanceOf[to] += amount;
    }
}
```

攻击场景：

```
1. Alice余额：0
2. Alice调用 transfer(Bob, 1)
3. balanceOf[Alice] = 0 - 1 = 2^256 - 1（天文数字！）
4. Alice现在拥有几乎无限的代币
```

后果：

- 代币供应量失控
- 经济模型崩溃
- 项目价值归零

3.3 真实案例

BeautyChain (BEC) - 2018年4月：

- 溢出漏洞导致代币凭空产生
- 市值蒸发数亿美元
- 多家交易所紧急暂停交易

SMT代币 - 2018年4月：

- 类似的溢出漏洞
- 攻击者铸造巨量代币
- 代币价格崩盘

教训：

- 溢出漏洞曾经非常普遍
- 必须使用SafeMath或Solidity 0.8.0+

3.4 Solidity 0.8.0的保护机制

版本对比：

Solidity 0.7.6 及以前：

```
pragma solidity ^0.7.6;

contract OldVersion {
    uint8 public value = 255;

    function increment() public {
        value++;  // 变成 0，不会报错！     }
}

// 需要手动使用SafeMath库：
using SafeMath for uint256;
value = value.add(1);  // 会检查溢出
```

Solidity 0.8.0 及以后：

```
pragma solidity ^0.8.19;

contract NewVersion {
    uint8 public value = 255;

    function increment() public {
        value++;  // 自动回滚，抛出Panic(0x11)     }
}

// 无需SafeMath：
value = value + 1;  // 自动安全 ```

**关键变化**：

- 自动检查溢出（默认行为）
- 溢出时自动回滚交易
- 抛出`Panic(0x11)`错误
- 无需SafeMath库

### 3.5 unchecked块的使用

**为什么需要unchecked**：

- Gas优化：溢出检查消耗额外Gas
- 确定安全的场景可以跳过检查

**使用示例**：
```

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract GasOptimized {
    uint256 public counter;

    /**
     * @notice 不安全的递增（不推荐）
     * @dev 可能溢出，虽然在这个场景下不太可能
     */
    function unsafeIncrement() public {
        unchecked {
            counter++;  // 不检查溢出
        }
    }

    /**
     * @notice 安全的循环（推荐）
     * @dev 循环计数器确定不会溢出，使用unchecked节省Gas
     */
    function safeLoop() public {
        // 循环从0到100，i确定不会溢出
        for (uint256 i = 0; i < 100; ) {
            // 处理逻辑...

            unchecked {
                i++;  // 确定不会溢出，节省Gas
            }
        }
    }
}
```

Gas对比：

- 带检查：约 1000 Gas
- unchecked：约 500 Gas
- 节省：约 50%

最佳实践：

1. **默认依赖自动检查**：
    - 不需要SafeMath
    - 编译器自动保护
    - 更清晰的代码
2. **谨慎使用unchecked**：
    - 只在确定安全时使用
    - 添加注释说明原因
    - 循环计数器是常见场景
3. **注意混合运算**：

    - 不同大小的类型转换
    - 先检查范围再转换
    - 使用显式类型转换
4. **测试边界情况**：
    - 最大值 + 1
    - 最小值 - 1
    - 类型转换边界

安全类型转换示例：

```
function safeCast(uint256 value) public pure returns (uint8) {
    require(value <= type(uint8).max, "Value too large");
    return uint8(value);
}
```

---

4. 权限控制漏洞（Access Control）

权限控制是智能合约安全的基础。如果关键函数缺少权限检查，任何人都可以调用，可能导致资金被盗或系统被破坏。

4.1 什么是权限控制漏洞

权限控制漏洞的定义：

权限控制漏洞是指合约中的关键函数缺少权限检查，导致未授权的用户可以调用这些函数，执行本不应该执行的操作。

常见场景：

1. **缺少onlyOwner修饰符**：
    - mint函数任何人都可以调用
    - 攻击者可以给自己铸造无限代币
2. **构造函数未初始化owner**：
    - owner是零地址
    - 无法执行管理操作
3. **权限检查逻辑错误**：
    - 检查条件写反
    - 权限验证被绕过

4.2 不安全的代码示例

缺少权限检查的代币合约：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

//  缺少权限检查的代币合约
contract UnsafeToken {
```

```
    mapping(address => uint256) public balances;

    /**
     * @notice 铸造函数（存在权限漏洞！）
     * @dev  危险：任何人都可以调用！
     */
    function mint(address to, uint256 amount) external {
        //  没有权限检查
        // 攻击者可以给自己铸造无限代币
        balances[to] += amount;
    }

    /**
     * @notice 销毁函数
     */
    function burn(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
    }
}
```

后果：

- 攻击者可以给自己铸造无限代币
- 代币价值归零
- 项目崩溃

4.3 安全的代码示例

正确的权限控制：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

//  正确的权限控制
contract SafeToken {
    // 记录合约所有者
    address public owner;

    mapping(address => uint256) public balances;

    /**
     * @notice onlyOwner修饰符
     * @dev 只有所有者可以调用
     */
    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }

    /**
```

```
     * @notice 构造函数：初始化owner
     * @dev  关键：在构造函数中初始化owner
     */
    constructor() {
        owner = msg.sender;  //  初始化owner
    }

    /**
     * @notice 铸造函数（安全）
     * @dev  使用onlyOwner修饰符进行权限检查
     */
    function mint(address to, uint256 amount)
        external
        onlyOwner  //  权限检查
    {
        balances[to] += amount;
    }

    /**
     * @notice 销毁函数
     * @dev 任何人都可以销毁自己的代币
     */
    function burn(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
    }

    /**
     * @notice 转移所有权
     * @dev 只有当前所有者可以转移
     */
    function transferOwnership(address newOwner) external onlyOwner {
        require(newOwner != address(0), "Invalid owner");
        owner = newOwner;
    }
}
```

4.4 Parity钱包漏洞案例分析

事件概况：

- **第一次攻击（2017年7月）**：损失约 $30,000,000
- **第二次冻结（2017年11月）**：冻结约 $150,000,000
- **原因**：`initWallet`函数可被任何人调用

漏洞代码分析：

```
// Parity钱包的initWallet函数（简化版）
contract WalletLibrary {
    address public owner;

    //  漏洞：initWallet可被任何人调用
```

```
    function initWallet(address[] _owners, uint _required, uint _daylimit) {
        // 如果owner已经设置，应该revert
        // 但攻击者可以在owner设置之前调用
        if (owner != address(0)) return;  // 只检查，不revert

        owner = msg.sender;  // 攻击者成为owner
        // ... 初始化其他参数
    }
}
```

攻击流程：

1. 攻击者调用`initWallet`
2. 攻击者成为owner
3. 攻击者提取所有资金

教训：

- 构造函数必须正确初始化权限
- 关键函数必须有权限检查
- 使用OpenZeppelin的Ownable

4.5 使用OpenZeppelin的Ownable

OpenZeppelin提供了经过审计的权限控制实现：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 导入OpenZeppelin的Ownable
import "@openzeppelin/contracts/access/Ownable.sol";

// 使用OpenZeppelin Ownable的代币合约
contract SecureToken is Ownable {
    mapping(address => uint256) public balances;

    /**
     * @notice 构造函数
     * @dev Ownable会自动将msg.sender设置为owner
     */
    constructor() Ownable() {
        // owner已经在Ownable的构造函数中设置
    }

    /**
     * @notice 铸造函数
     * @dev 使用onlyOwner修饰符
     */
    function mint(address to, uint256 amount) external onlyOwner {
        balances[to] += amount;
    }
```

```
    /**
     * @notice 销毁函数
     */
    function burn(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
    }
}
```

OpenZeppelin Ownable的优势：

- 经过充分审计
- 提供`transferOwnership`和`renounceOwnership`
- 事件记录所有权变更
- 开箱即用

4.6 使用AccessControl实现角色管理

对于更复杂的权限需求，可以使用OpenZeppelin的AccessControl：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "@openzeppelin/contracts/access/AccessControl.sol";

contract SecureTokenWithRoles is AccessControl {
    // 定义角色
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant BURNER_ROLE = keccak256("BURNER_ROLE");

    mapping(address => uint256) public balances;

    constructor() {
        // 设置默认管理员
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);

        // 给部署者授予MINTER和BURNER角色
        _setupRole(MINTER_ROLE, msg.sender);
        _setupRole(BURNER_ROLE, msg.sender);
    }

    /**
     * @notice 铸造函数
     * @dev 只有MINTER角色可以调用
     */
    function mint(address to, uint256 amount) external onlyRole(MINTER_ROLE) {
        balances[to] += amount;
    }

    /**
     * @notice 销毁函数
```

```
     * @dev 只有BURNER角色可以调用
     */
    function burn(address from, uint256 amount) external onlyRole(BURNER_ROLE) {
        require(balances[from] >= amount, "Insufficient balance");
        balances[from] -= amount;
    }
}
```

AccessControl的优势：

- 支持多个角色
- 角色可以授予和撤销
- 更灵活的权限管理
- 适合复杂的权限需求

---

5. 拒绝服务攻击（DoS）

拒绝服务攻击（Denial of Service，DoS）是指攻击者通过消耗Gas、阻塞状态等方式，使合约的某些功能无法正常使用。

5.1 DoS攻击的类型

DoS攻击主要有以下几种形式：

1. **Gas耗尽攻击**：
    - 循环处理大量数据
    - 超过Gas限制无法执行
    - 导致功能完全不可用
2. **状态阻塞攻击**：
    - 占用关键状态
    - 阻止其他用户操作
    - 使系统陷入死锁
3. **外部依赖攻击**：
    - 依赖外部合约
    - 外部合约失败导致无法使用
    - 单点故障

5.2 Gas耗尽攻击示例

不安全的奖励分配合约：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

//  容易受DoS攻击的奖励合约
contract UnsafeReward {
    // 存储所有用户地址
    address[] public users;
```

```
    // 记录每个用户是否已领取奖励
    mapping(address => bool) public hasClaimed;

    /**
     * @notice 分配奖励（存在DoS漏洞！）
     * @dev  危险：如果用户太多，Gas耗尽!
     */
    function distributeRewards() external {
        //  危险：循环处理所有用户
        // 如果users数组很大，Gas消耗会超过区块Gas限制
        // 导致函数永远无法执行成功
        for (uint256 i = 0; i < users.length; i++) {
            address user = users[i];

            // 如果用户还没领取，发送奖励
            if (!hasClaimed[user]) {
                hasClaimed[user] = true;
                // 向用户转账
                payable(user).transfer(1 ether);
            }
        }
    }

    /**
     * @notice 添加用户
     */
    function addUser(address user) external {
        users.push(user);
    }
}
```

攻击场景：

1. 攻击者调用`addUser`添加大量地址（例如1000个）
2. 当调用`distributeRewards`时，需要循环1000次
3. Gas消耗超过区块Gas限制（例如15,000,000）
4. 交易失败，奖励永远无法分配

问题分析：

- 无界循环：`users.length`可能非常大
- 每次循环都有外部调用（`transfer`）
- 外部调用消耗大量Gas
- 总Gas消耗可能超过区块限制

5.3 拉取模式（Pull Pattern）防御

安全的奖励分配合约：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;
```

```
//  使用拉取模式的奖励合约
contract SafeReward {
    // 记录每个用户的奖励金额
    mapping(address => uint256) public rewards;

    /**
     * @notice 设置用户奖励
     * @dev 只更新状态，不进行外部调用
     */
    function setReward(address user, uint256 amount) external {
        rewards[user] = amount;
    }

    /**
     * @notice 用户主动领取奖励（拉取模式）
     * @dev  安全：用户主动调用，Gas由用户承担
     */
    function claimReward() external {
        // 检查用户是否有待领取的奖励
        uint256 amount = rewards[msg.sender];
        require(amount > 0, "No reward");

        // 清零奖励（防止重复领取）
        rewards[msg.sender] = 0;

        // 向用户转账
        // 由于是用户主动调用，Gas由用户承担
        // 即使Gas消耗较高，也不会影响其他用户
        payable(msg.sender).transfer(amount);
    }
}
```

拉取模式的优势：

1. **Gas由用户承担**：
   - 用户主动调用`claimReward`
   - Gas消耗由用户支付
   - 不会因为Gas耗尽导致功能不可用
2. **无界循环问题解决**：
   - 不需要循环处理所有用户
   - 每个用户独立领取
   - 不受用户数量限制
3. **更好的用户体验**：
   - 用户可以选择何时领取
   - 可以分批领取
   - 更灵活

5.4 其他防御方法

1. 限制数组和循环大小：

```
contract LimitedLoop {
    address[] public users;
    uint256 public constant MAX_BATCH_SIZE = 100;

    /**
     * @notice 批量处理（限制大小）
     * @dev 每次最多处理100个用户
     */
    function batchProcess(uint256 startIndex, uint256 endIndex) external {
        require(endIndex - startIndex <= MAX_BATCH_SIZE, "Batch too large");
        require(endIndex <= users.length, "Index out of range");

        for (uint256 i = startIndex; i < endIndex; i++) {
            // 处理逻辑...
        }
    }
}
```

2. 使用分页处理：

```
contract PaginatedProcess {
    address[] public users;
    uint256 public constant PAGE_SIZE = 50;
    uint256 public processedCount;

    /**
     * @notice 分页处理
     * @dev 每次处理一页，可以多次调用
     */
    function processNextPage() external {
        uint256 endIndex = processedCount + PAGE_SIZE;
        if (endIndex > users.length) {
            endIndex = users.length;
        }

        for (uint256 i = processedCount; i < endIndex; i++) {
            // 处理逻辑...
        }

        processedCount = endIndex;
    }
}
```

3. 添加超时和取消机制：

```
contract TimeoutProcess {
    struct Process {
        uint256 startTime;
```

```
        bool completed;
    }

    mapping(uint256 => Process) public processes;
    uint256 public constant TIMEOUT = 1 hours;

    /**
     * @notice 处理任务
     */
    function process(uint256 taskId) external {
        require(!processes[taskId].completed, "Already completed");
        require(
            block.timestamp - processes[taskId].startTime < TIMEOUT,
            "Timeout"
        );

        // 处理逻辑...
        processes[taskId].completed = true;
    }

    /**
     * @notice 取消超时任务
     */
    function cancelTimeout(uint256 taskId) external {
        require(
            block.timestamp - processes[taskId].startTime >= TIMEOUT,
            "Not timeout"
        );
        // 取消逻辑...
    }
}
```

防御策略总结：

- 使用拉取模式而非推送模式
- 限制数组和循环大小
- 使用分页处理大量数据
- 添加超时和取消机制
- 减少对外部依赖

---

6. 前端运行攻击（Front-Running）

前端运行攻击是区块链特有的安全风险。由于所有交易都在mempool中公开可见，攻击者可以观察有利可图的交易，然后提交Gas更高的相同交易，让自己的交易先被执行。

6.1 什么是前端运行攻击

前端运行攻击的定义：

前端运行攻击（Front-Running Attack）是指攻击者观察到mempool中的有利可图交易后，提交Gas更高的相同交易，让自己的交易先被执行，从而获得利益。

攻击流程：

```
用户提交交易
    ↓
交易进入mempool（公开可见）
    ↓
攻击者观察到有利可图的交易
    ↓
攻击者提交Gas更高的相同交易      ↓
攻击者的交易先被执行      ↓
用户的交易后执行（已无利可图）```

**为什么叫"前端运行"**:

攻击者的交易"跑在"用户交易的前面，先被执行。

### 6.2 攻击场景示例

**场景1：DEX价格套利**:
```

1. 用户提交交易：用100 ETH购买代币A
2. 攻击者观察到：这个交易会推高代币A的价格
3. 攻击者抢先：用更高的Gas购买代币A
4. 攻击者获利：代币A价格上涨后卖出
5. 用户损失：以更高的价格购买代币A

```
**场景2：NFT抢购**:
```

1. 用户提交交易：购买限量NFT（价格100 ETH）
2. 攻击者观察到：这是最后一个NFT
3. 攻击者抢先：用更高的Gas购买
4. 攻击者获得：最后一个NFT
5. 用户失败：交易被拒绝（NFT已售完）

```
### 6.3 Commit-Reveal防御方案

Commit-Reveal方案通过分两阶段提交来防止前端运行：

**阶段1：提交哈希（Commit）**:
- 用户提交交易的哈希值
- 真实交易内容被隐藏
- 攻击者无法知道交易详情
```

阶段2：提交真实交易（Reveal）：
- 用户提交真实的交易内容
- 验证哈希是否匹配
- 执行交易

实现示例：

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

//  Commit-Reveal投票合约
contract CommitRevealVoting {
    // 记录每个用户的提交哈希
    mapping(address => bytes32) public commits;

    // 记录每个选项的投票数
    mapping(uint256 => uint256) public votes;

    // 投票阶段：1=提交阶段，2=揭示阶段
    uint256 public phase = 1;

    /**
     * @notice 阶段1：提交哈希（隐藏真实投票）
     * @param hash 投票内容和随机数的哈希
     * @dev 攻击者无法知道真实投票内容
     */
    function commit(bytes32 hash) external {
        require(phase == 1, "Not commit phase");
        commits[msg.sender] = hash;
    }

    /**
     * @notice 阶段2：提交真实投票（验证哈希）
     * @param choice 投票选项
     * @param nonce 随机数（用于生成哈希）
     * @dev 验证哈希是否匹配，然后记录投票
     */
    function reveal(uint256 choice, uint256 nonce) external {
        require(phase == 2, "Not reveal phase");

        // 计算哈希：choice + nonce + msg.sender
        bytes32 hash = keccak256(
            abi.encodePacked(choice, nonce, msg.sender)
        );

        // 验证哈希是否匹配
        require(hash == commits[msg.sender], "Invalid commit");

        // 清零提交（防止重复揭示）
        commits[msg.sender] = bytes32(0);
```

```
        // 记录投票
        votes[choice]++;
    }

    /**
     * @notice 切换到揭示阶段
     * @dev 只有管理员可以调用
     */
    function startRevealPhase() external {
        phase = 2;
    }
}
```

Commit-Reveal的优势：

- 隐藏真实交易内容
- 攻击者无法提前知道交易详情
- 防止前端运行

Commit-Reveal的缺点：

- 需要两阶段提交
- 用户体验较差
- 需要额外的Gas消耗

6.4 其他防御方法

1. 使用私有交易池（如Flashbots）：

- 交易不进入公共mempool
- 直接提交给矿工
- 攻击者无法观察到交易

2. 设置价格滑点限制：

```
contract SlippageProtection {
    function swap(uint256 amountIn, uint256 minAmountOut) external {
        uint256 amountOut = calculateSwap(amountIn);

        // 检查滑点是否在允许范围内
        require(amountOut >= minAmountOut, "Slippage too high");

        // 执行交换...
    }
}
```

3. 批量拍卖机制：

- 所有交易在同一区块执行
- 使用统一价格
- 防止交易顺序影响价格

4. 最小延迟机制：

- 交易提交后延迟执行
- 给其他用户反应时间
- 减少前端运行的优势

防御策略总结：

- Commit-Reveal方案（分两阶段）
- 使用私有交易池（如Flashbots）
- 设置价格滑点限制
- 批量拍卖机制
- 最小延迟机制

---

7. 安全检查清单

在部署合约之前，使用系统化的安全检查清单可以避免遗漏关键安全问题。以下是完整的安全检查清单。

7.1 重入攻击防护

检查项：

- [ ] 使用CEI模式（Checks-Effects-Interactions）
- [ ] 关键函数使用重入锁（ReentrancyGuard）
- [ ] 避免循环中的外部调用
- [ ] 状态在外部调用前更新

示例检查：

```
//  正确：遵循CEI模式
function withdraw() external {
    // Checks
    require(balances[msg.sender] > 0, "No balance");

    // Effects
    balances[msg.sender] = 0;

    // Interactions
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
}
```

7.2 权限控制

检查项：

- [ ] 所有关键函数有权限检查
- [ ] 使用onlyOwner修饰符
- [ ] 构造函数正确初始化owner

- [ ] 使用OpenZeppelin的Ownable或AccessControl

示例检查：

```
//  正确：使用onlyOwner
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
}

function mint(address to, uint256 amount) external onlyOwner {
    // ...
}
```

7.3 输入验证

检查项：

- [ ] 检查地址不为零地址
- [ ] 检查数值范围
- [ ] 检查数组边界
- [ ] 验证所有外部输入

示例检查：

```
//  正确：验证输入
function transfer(address to, uint256 amount) external {
    require(to != address(0), "Invalid recipient");
    require(amount > 0, "Invalid amount");
    require(balances[msg.sender] >= amount, "Insufficient balance");
    // ...
}
```

7.4 其他安全检查

检查项：

- [ ] 使用Solidity 0.8.0+（防止整数溢出）
- [ ] 避免无界循环
- [ ] 使用拉取模式分配资金
- [ ] 检查外部调用返回值
- [ ] 添加超时和取消机制
- [ ] 敏感操作使用Commit-Reveal
- [ ] 记录所有重要操作（emit事件）
- [ ] 编写完整的测试
- [ ] 进行专业审计

7.5 部署前检查

最后检查：

1. **代码审查**：
   - 团队内部代码审查
   - 使用自动化工具扫描（如Slither）
   - 检查所有警告和错误
2. **测试覆盖**：
   - 单元测试覆盖所有函数
   - 集成测试覆盖主要流程
   - 边界条件测试
   - 攻击场景测试
3. **审计**：
   - 专业安全公司审计
   - 社区代码审查
   - Bug赏金计划
4. **部署准备**：
   - 确认所有检查项都通过
   - 准备应急响应计划
   - 设置监控和告警

记住：部署前必须全部通过检查！安全是第一要务。

---

8. 实践练习

理论学习之后，实践是巩固知识的最好方式。以下是不同难度的练习题目。

8.1 练习1：修复重入漏洞（二星难度）

任务：找到提供的不安全Vault合约，修复重入漏洞。

要求：

1. 使用CEI模式修复
2. 使用重入锁修复
3. 对比两种方法
4. 编写测试验证安全性

参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

// 使用CEI模式修复
contract SecureVaultCEI {
    mapping(address => uint256) public balances;
```

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

    function withdraw() external {
        // Checks
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        // Effects
        balances[msg.sender] = 0;

        // Interactions
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}

// 使用重入锁修复
contract SecureVaultLock {
    mapping(address => uint256) public balances;
    bool private locked;

    modifier noReentrant() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }

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

    function withdraw() external noReentrant {
        uint256 amount = balances[msg.sender];
        require(amount > 0, "No balance");

        balances[msg.sender] = 0;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

8.2 练习2：实现权限控制（二星难度）

任务：为代币合约添加完整的权限控制。

要求：

1. 使用OpenZeppelin的Ownable
2. 实现mint和burn函数

3. 只有owner可以mint
4. 任何人可以burn自己的代币
5. 添加权限转移功能

参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

import "@openzeppelin/contracts/access/Ownable.sol";

contract SecureToken is Ownable {
    mapping(address => uint256) public balances;

    constructor() Ownable() {}

    function mint(address to, uint256 amount) external onlyOwner {
        balances[to] += amount;
    }

    function burn(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
    }
}
```

8.3 练习3：防御DoS攻击（三星难度）

任务：实现安全的奖励分配合约。

要求：

1. 使用拉取模式
2. 避免Gas限制问题
3. 添加超时机制
4. 支持批量处理

参考答案：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract SafeRewardDistribution {
    mapping(address => uint256) public rewards;
    mapping(address => uint256) public claimDeadline;
    uint256 public constant CLAIM_PERIOD = 30 days;

    function setReward(address user, uint256 amount) external {
        rewards[user] = amount;
        claimDeadline[user] = block.timestamp + CLAIM_PERIOD;
    }
```

```
    function claimReward() external {
        uint256 amount = rewards[msg.sender];
        require(amount > 0, "No reward");
        require(block.timestamp <= claimDeadline[msg.sender], "Expired");

        rewards[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

8.4 练习4：安全审查（难度）

任务：使用安全检查清单审查一个现有合约。

要求：

1. 逐项检查清单
2. 发现并记录所有安全问题
3. 提供修复建议
4. 编写审查报告

提示：

- 可以从GitHub上找一个简单的DeFi协议
- 使用Slither等工具辅助分析
- 重点关注重入、权限、溢出等问题

---

9. 学习检查清单

完成本课后，你应该能够：

重入攻击：

- [ ] 理解重入攻击的原理
- [ ] 知道如何识别重入漏洞
- [ ] 会使用CEI模式防御
- [ ] 会使用重入锁防御
- [ ] 了解The DAO攻击案例

整数溢出：

- [ ] 理解溢出和下溢的概念
- [ ] 知道Solidity 0.8.0的保护机制
- [ ] 会正确使用unchecked块
- [ ] 了解历史溢出漏洞案例

权限控制：

- [ ] 理解权限控制的重要性
- [ ] 会实现onlyOwner修饰符
- [ ] 会使用OpenZeppelin的Ownable
- [ ] 了解Parity钱包漏洞案例

DoS攻击：

- [ ] 理解DoS攻击的类型
- [ ] 知道Gas耗尽攻击的原理
- [ ] 会使用拉取模式防御
- [ ] 会限制循环大小

前端运行：

- [ ] 理解前端运行攻击的原理
- [ ] 知道Commit-Reveal方案
- [ ] 了解其他防御方法
- [ ] 理解MEV的概念

安全检查：

- [ ] 会使用安全检查清单
- [ ] 知道部署前的检查流程
- [ ] 了解代码审计的重要性

---

10. 总结

智能合约安全是开发中最重要的一环。通过本课的学习，你应该已经掌握了：

1. **重入攻击**：
   - 最危险的漏洞之一
   - 使用CEI模式和重入锁防御
   - The DAO攻击是典型案例
2. **整数溢出**：
   - Solidity 0.8.0已内置保护
   - 谨慎使用unchecked块
   - 多个代币合约曾受此漏洞影响
3. **权限控制**：
   - 基础但致命
   - 使用OpenZeppelin的Ownable
   - Parity钱包漏洞是典型案例
4. **DoS攻击**：
   - Gas耗尽和状态阻塞
   - 使用拉取模式防御
   - 限制循环和数组大小

5. **前端运行**：
    - 区块链特有的风险
    - Commit-Reveal方案可以防御
    - MEV是更大的问题

关键要点：

- 安全是第一要务
    - 智能合约一旦部署无法修改
    - 漏洞可能导致不可挽回的损失
- 使用经过审计的库
    - OpenZeppelin是行业标准
    - 不要重复造轮子
- 🔍 系统化安全审查
    - 使用安全检查清单
    - 不遗漏任何要点
- 📚 持续学习
    - 安全威胁不断演变
    - 关注最新漏洞和防御方法

核心知识点回顾：

漏洞类型	主要危害	防御方法	真实案例
重入攻击	资金被掏空	CEI模式、重入锁	The DAO ($60M)
整数溢出	数值异常、代币增发	Solidity 0.8+、避免unchecked	多个代币合约
权限控制	未授权操作、资金被盗	onlyOwner、AccessControl	Parity钱包 ($150M)
拒绝服务	功能不可用、Gas耗尽	拉取模式、限制循环	多个DeFi协议
前端运行	交易被抢跑、MEV攻击	Commit-Reveal、私有交易池	DEX交易、NFT抢购

记住：安全不是一次性的任务，而是持续的过程。每写一行代码，都要问自己："这里安全吗？攻击者会怎么利用？"

你的代码可能管理数百万美元的资金，安全意识和技能将伴随你的整个职业生涯！