OWASP 中国
Open Worldwide Application Security Project

OWASP
智能合约十大风险

OWASP Smart Contract Top 10

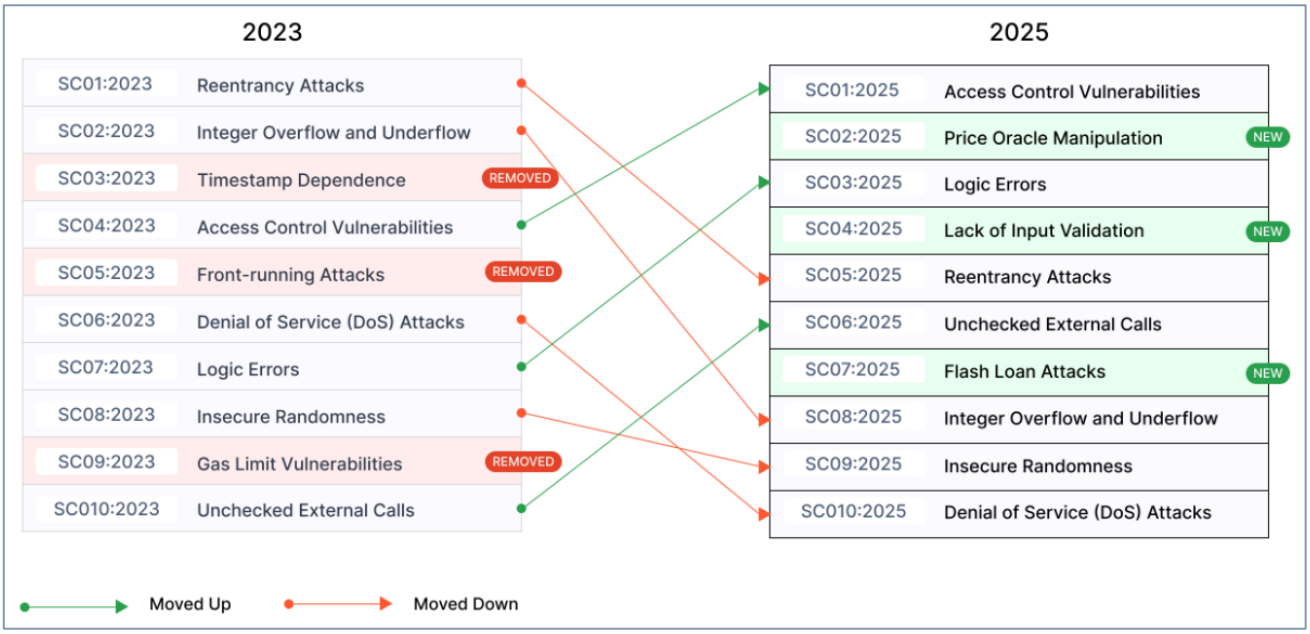
目录

1 关于智能合约十大安全风险

OWASP 智能合约十大安全风险（2025）是一份标准安全意识文档，旨在为 Web3 开发者和安全团队提供关于智能合约中最常见的十大漏洞的深入解析。

它作为参考指南，确保智能合约能够抵御近年来被利用或发现的最关键安全漏洞。智能合约十大安全风险可与其他智能合约安全项目结合使用，以实现全面的风险覆盖。欲了解更多关于 OWASP 智能合约安全项目的详细信息，请访问 scs.owasp.org。

1.1 与上一版本的变更



1.1.1 Top 10

SC01:2025	访问控制漏洞	Access Control Vulnerabilities
SC02:2025	价格预言机操纵	Price Oracle Manipulation
SC03:2025	逻辑错误	Logic Errors
SC04:2025	输入验证不足	Lack of Input Validation
SC05:2025	可重入攻击	Reentrancy Attacks
SC06:2025	未检查的外部调用	Unchecked External Calls
SC07:2025	闪电贷攻击	Flash Loan Attacks
SC08:2025	整数溢出与下溢	Integer Overflow and Underflow
SC09:2025	不安全的随机数	Insecure Randomness
SC10:2025	拒绝服务（DoS）攻击	Denial of Service (DoS) Attacks

1.1.2 OWASP 智能合约十大安全风险（2025）概述

编号	风险类别
SC01 - 访问控制漏洞	访问控制机制不当，可能使未经授权的用户获取或篡改智能合约的数据和功能。当权限检查缺失或设计不严谨时，可能导致合约遭受攻击，甚至资产被非法操控。
SC02 - 价格预言机操纵	智能合约依赖外部数据，攻击者可通过篡改或控制预言机数据影响合约逻辑，从而引发市场操纵、经济损失或系统不稳定。
SC03 - 逻辑漏洞	业务逻辑缺陷可能导致合约行为偏离预期，如奖励分配错误、代币铸造异常或借贷规则漏洞，可能引发资产损失或经济模型失衡。
SC04 - 输入验证不足	由于缺乏对输入数据的严格校验，攻击者可提交恶意或异常输入，破坏合约逻辑，导致安全隐患，甚至触发不可逆的合约错误。
SC05 - 可重入攻击	攻击者在合约执行未完成前反复调用关键函数，导致合约状态异常，如重复提款或错误的资金转移，最终造成资产损失。
SC06 - 未受保护的外部调用	智能合约未正确处理外部调用的返回结果，可能在调用失败时仍继续执行，导致逻辑异常、权限绕过或资金丢失。
SC07 - 闪电贷攻击	利用闪电贷在单个交易内执行多个操作，攻击者可以操纵市场价格、抽干流动性或破坏合约业务逻辑，造成巨大经济损失。
SC08 - 整数溢出与下溢	计算过程中数值超出整数范围可能引发错误，如代币供应异常、数值绕回或权限绕过，最终影响合约安全性。
SC09 - 不安全的随机数	由于区块链的确定性特性，随机数生成往往可预测或被操控，可能被用于操纵抽奖、公平分配或博弈系统，造成不公平竞争。
SC10 - 拒绝服务（DoS）攻击	攻击者通过消耗智能合约资源，如高计算成本的循环或恶意函数调用，使合约陷入不可用状态，影响正常业务运行。

1.2 数据来源

1.2.1 SolidityScan 的 Web3HackHub

为识别和验证 OWASP 智能合约十大安全风险，我们综合参考了多个权威数据来源，其中 SolidityScan 的 Web3HackHub（2024） 提供了关键支持。该资源汇总了区块链领域的安全事件，深入分析攻击方式、资金损失及行业趋势，为智能合约安全研究提供了翔实的数据基础。

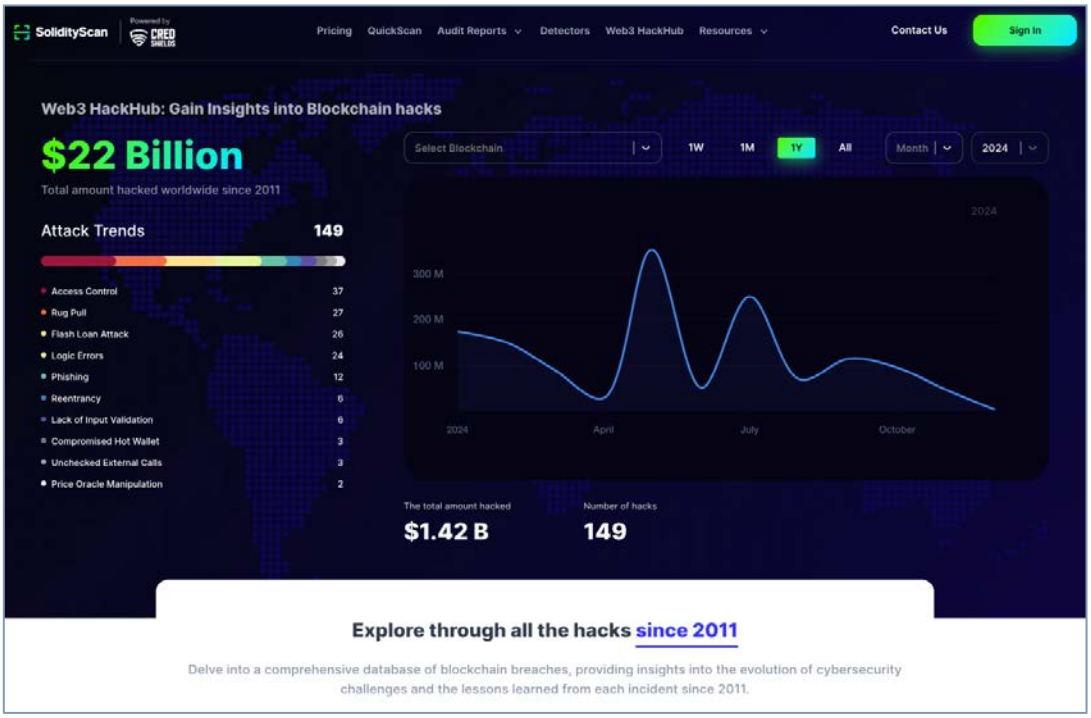
Web3HackHub 记录了 2011 年以来 的安全事件，使研究人员能够追踪攻击手段的演变，分析漏洞利用的日益复杂化，并从每次事件中总结出关键经验。

2024 年 Web3HackHub 数据摘要

年度总经济损失：2024 年共记录 149 起安全事件，累计损失 14.2 亿美元。

主要攻击方式（按发生频率及总损失排名）：

- 访问控制漏洞 - 损失 9.532 亿美元
- 逻辑漏洞 - 损失 6380 万美元
- 可重入攻击 - 损失 3570 万美元
- 闪电贷攻击 - 损失 3380 万美元
- 输入验证不足 - 损失 1460 万美元
- 价格预言机操纵 - 损失 880 万美元
- 未受保护的外部调用 - 损失 55.07 万美元



1.2.2 其他数据来源

除了 SolidityScan 的 Web3HackHub，Peter Kacherginsky 的《2024 年 DeFi 十大攻击向量》 也提供了重要的分析视角，为制定 OWASP 智能合约十大安全风险（2025） 提供了有力的数据支持。Peter 的研究深入剖析了 DeFi 领域不断演化的安全威胁，使本次排名更加贴合真实攻击案例，并反映最新的安全趋势。

通过整合 SolidityScan 的 Web3HackHub 与 Kacherginsky 的《2024 年 DeFi 十大攻击向量》 数据，我们得以全面评估智能合约安全威胁，并为 2025 年排名提供科学依据。

在分析 149 起安全事件 后，我们还参考了 Immunefi《2024 年加密货币损失报告》，该报告详细记录了 去中心化生态系统超过 14.2 亿美元的财务损失。基于这些数据，OWASP 制定了 2025 年智能合约十大安全风险，重点关注区块链与智能合约生态系统中最严重的安全漏洞。

2 许可协议

3 项目 Leaders

- Jinson Varghese Behanan (Twitter: @JinsonCyberSec)
- Shashank (Twitter: @cyberboyIndia)

中文项目组

项目组组长： 张坤（破天）

项目组成员： 刘畅（玄道）

SC01:2025 访问控制不当

描述

访问控制漏洞是指合约未能正确限制用户权限，导致未经授权的用户可以访问或修改合约的数据或功能。这类漏洞通常源于智能合约代码未能充分实施权限管理，导致关键操作（如代币铸造、治理投票、资金提取、合约暂停与升级、所有权变更等）暴露给非授权用户。

潜在影响

- 攻击者可未经授权访问智能合约中关键功能和数据，如篡改数据或调用受保护功能，破坏合约完整性。
- 资金或资产可能被恶意转移或销毁，导致用户和利益相关者遭受严重财务损失。

修复方案

- 确保初始化函数只能被调用一次，并且仅允许授权实体调用。
- 采用成熟的访问控制模式来管理权限，如 Ownable 或基于角色的访问控制（RBAC），确保只能由授权用户访问敏感功能。在关键函数中添加适当的权限控制修饰符，如 onlyOwner 或自定义角色管理机制。

真实案例

1. HospoWise Hack : A Comprehensive Hack Analysis
2. LAND NFT Hack : A Comprehensive Hack Analysis

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_AccessControl {
    mapping(address => uint256) public balances;

    // Burn function with no access control（该销毁函数没有权限控制，任何人都可以调用）
    function burn(address account, uint256 amount) public {
        _burn(account, amount);
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

// 引入 OpenZeppelin 的 Ownable 合约，管理所有权
import "@openzeppelin/contracts/access/Ownable.sol";

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

    //  Burn function with proper access control, only accessible by the contract owner（只有合约所有者才能执行销毁操作）
    function burn(address account, uint256 amount) public onlyOwner {
        _burn(account, amount);
    }
}
```

SC02:2025 价格预言机操纵

描述

价格预言机操纵是一种严重的智能合约漏洞，主要发生在依赖外部数据源（预言机）获取价格或其他关键信息的合约中。在 去中心化金融（DeFi） 领域，预言机负责向智能合约提供真实世界的数据，如资产价格。然而，如果预言机提供的数据被篡改，合约逻辑可能因此失真，导致一系列灾难性后果，例如：非法超额借款、过度杠杆利用、流动性池被抽干。因此，在设计依赖预言机的智能合约时，必须采取严格的安全防护措施，以防止此类攻击。

潜在影响

- **恶意抬高资产价格：**攻击者可通过操控预言机数据，使某项资产价格虚高，从而超额借款，获取不应得的资金。
- **清算风险加剧：**被操控的价格可能导致错误的抵押价值计算，导致正常用户的资产被清算，造成重大损失。
- **合约资金池被掏空：**一旦预言机遭受攻击，攻击者可能利用操控后的数据进行套利交易，甚至令合约流动性耗尽，导致整个协议破产。

修复方案

- **使用多个独立的预言机数据源：**通过聚合多个去中心化预言机的数据，降低单一预言机被操控的风险。
- **设定价格波动阈值：**对接收到的价格数据设定最小值和最大值，防止极端价格变动影响合约逻辑。
- **引入时间锁机制：**对价格更新引入时间延迟，防止攻击者在极短时间内操纵价格并立即执行恶意交易。
- **使用加密证明机制：**确保从预言机接收到的数据具有加密签名，只有可信数据源的签名才能被接受。

真实案例

1. Polter Finance Hack Analysis
2. BonqDAO Protocol : A Comprehensive Hack Analysis

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IPriceFeed {
    function getLatestPrice() external view returns (int);
}
```

```
contract PriceOracleManipulation {
    address public owner;
    IPriceFeed public priceFeed;

    constructor(address _priceFeed) {
        owner = msg.sender;
        priceFeed = IPriceFeed(_priceFeed);
    }

    function borrow(uint256 amount) public {
        int price = priceFeed.getLatestPrice();
        require(price > 0, "Price must be positive");

        // Vulnerability: No validation or protection against price manipulation（漏洞：未对价格数据进行验证或防范操纵）
        uint256 collateralValue = uint256(price) * amount;

        // If an attacker manipulates the oracle, they could borrow more than they should（攻击者若操纵预言机价格，可能借出远超应得的资金）
    }

    function repay(uint256 amount) public {
        // Repayment logic（还款逻辑）
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IPriceFeed {
    function getLatestPrice() external view returns (int);
}

contract PriceOracleManipulation {
    address public owner;
    IPriceFeed public priceFeed;
```

```
    int public minPrice = 1000; // Set minimum acceptable price（设定最低可接受价格）
    int public maxPrice = 2000; // Set maximum acceptable price（设定最高可接受价格）

    constructor(address _priceFeed) {
        owner = msg.sender;
        priceFeed = IPriceFeed(_priceFeed);
    }

    function borrow(uint256 amount) public {
        int price = priceFeed.getLatestPrice();
        require(price > 0 && price >= minPrice && price <= maxPrice, "Price manipulation detected");

        uint256 collateralValue = uint256(price) * amount;

        // Borrow logic based on valid price（仅在价格数据通过验证时，执行借款逻辑）
    }

    function repay(uint256 amount) public {
        // Repayment logic（还款逻辑）
    }
}
```

SC03:2025 逻辑错误

描述

逻辑错误（也称业务逻辑漏洞）是智能合约中隐蔽且危险的缺陷，通常源于代码未能正确实现预期的行为。这类漏洞可能表现为奖励分配计算错误、代币铸造机制异常，或借贷逻辑计算失误等。由于逻辑错误通常隐藏在代码逻辑深处，难以被察觉，往往只有在特定条件下才会暴露，造成严重影响。

常见逻辑错误示例

1. **奖励分配错误：**计算不准确，导致奖励分配不公，影响公平性。
2. **代币铸造机制缺陷：**未正确验证或存在错误的铸造逻辑，可能导致无限制或非预期的代币生成。
3. **借贷池失衡：**存款与提款记录错误，导致资金池数据不一致。

潜在影响

逻辑错误可能导致智能合约表现异常，甚至完全无法使用。这些错误可能引发以下后果：

- **资金损失：**奖励分配或借贷池资产计算错误可能导致资金被滥用或流失。
- **代币供应失衡：**无控制的代币铸造可能导致供应量激增，破坏市场信任和代币价值。
- **合约功能失效：**智能合约可能因逻辑错误而无法正常执行预期功能，影响系统的稳定性和用户体验。

这些后果可能给用户和利益相关方带来巨大的财务和运营损失。

修复方案

- **完善测试覆盖率：**编写全面的测试用例，确保所有可能的业务逻辑场景均经过验证。
- **严格代码审查与安全审计：**定期进行代码审核，发现并修复潜在的逻辑漏洞。
- **文档化业务逻辑：**清晰记录每个函数的预期行为，并确保实现与设计保持一致。
- **引入安全机制：**
  - 使用安全数学库（SafeMath）避免计算错误。
  - 对代币铸造逻辑添加严格验证机制，防止滥发代币。
  - 采用可审计的奖励分配算法，确保公平性。

真实案例

1. Level Finance Hack ：A Comprehensive Hack Analysis
2. BNO Hack ：A Comprehensive Hack Analysis

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_LogicErrors {
    mapping(address => uint256) public userBalances;
    uint256 public totalLendingPool;

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

    function withdraw(uint256 amount) public {
        require(userBalances[msg.sender] >= amount, "Insufficient balance");

        // Faulty calculation: Incorrectly reducing the user's balance without updating the total lending pool（逻辑错误：未正确更新总借贷池余额，导致资金数据不匹配）
        userBalances[msg.sender] -= amount;

        // This should update the total lending pool, but it's omitted here.（这里应该更新总借贷池余额，但被忽略了）

        payable(msg.sender).transfer(amount);
    }

    function mintReward(address to, uint256 rewardAmount) public {
        // Faulty minting logic: Reward amount not validated（逻辑错误：未验证奖励金额是否合理）
        userBalances[to] += rewardAmount;
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_LogicErrors {
    mapping(address => uint256) public userBalances;
    uint256 public totalLendingPool;

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

    function withdraw(uint256 amount) public {
        require(userBalances[msg.sender] >= amount, "Insufficient balance");

        // Correctly reducing the user's balance and updating the total lending pool（纠正逻辑错误：在减少用户余额的同时正确更新借贷池总余额）
        userBalances[msg.sender] -= amount;
        totalLendingPool -= amount;

        payable(msg.sender).transfer(amount);
    }

    function mintReward(address to, uint256 rewardAmount) public {
        require(rewardAmount > 0, "Reward amount must be positive");

        // Safeguarded minting logic（增加验证机制，确保奖励分配合理）
        userBalances[to] += rewardAmount;
    }
}
```

SC04:2025 输入验证不足

描述

输入验证是确保智能合约仅处理有效且符合预期数据的关键安全措施。如果合约未能严格验证外部输入，攻击者可以利用这一漏洞篡改合约逻辑、绕过权限控制，甚至引入恶意数据，导致不可预测的安全风险。例如，如果合约默认信任用户输入而未进行验证，攻击者可以操纵数据，导致资金损失或系统异常。

潜在影响

- **资金流失：**攻击者可利用漏洞伪造余额，非法获取超出其应得份额的合约中的资产。可能会将合同余额完全抽空。
- **合约状态被污染：**未受控制的输入可能导致状态变量错误更新，影响合约的可靠性和安全性。
- **未经授权的操作：**攻击者可利用不受限制的输入，执行非预期交易或操作，危及用户和系统的安全性。

修复方案

- **确保输入符合预期类型：**验证输入数据的格式、类型和范围，防止意外值的注入。
- **限制输入范围：**设定合理的参数边界，防止超出预期的异常数据被提交。
- **实施访问控制：**确保只有授权实体可以调用关键函数，避免恶意用户篡改关键数据。
- **验证输入结构：**对于地址或字符串等输入，检查格式是否符合预期，避免非法数据写入。
- **异常处理：**当输入验证失败时，应立即终止执行，并提供清晰的错误信息。

真实案例

1. Convergence Finance : A Comprehensive Hack Analysis
2. Socket Gateway : A Comprehensive Hack Analysis

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_LackOfInputValidation {
    mapping(address => uint256) public balances;
```

```
    function setBalance(address user, uint256 amount) public {
        // The function allows anyone to set arbitrary balances for any user without validation.（该函数允许任何人为任意用户设定余额，且未进行任何输入验证）
        balances[user] = amount;
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract LackOfInputValidation {
    mapping(address => uint256) public balances;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Caller is not authorized");
        _;
    }

    function setBalance(address user, uint256 amount) public onlyOwner {
        require(user != address(0), "Invalid address");
        balances[user] = amount;
    }
}
```

SC05:2025 可重入攻击

描述

可重入攻击是一种常见的智能合约漏洞，发生在合约在更新自身状态之前进行外部调用时。恶意合约可以在外部调用返回之前递归调用原函数，反复执行某些操作（如提款），从而绕过余额检查，多次提取资金。攻击者可以利用这一漏洞，反复调用提款函数，直至合约资金被全部提取。

潜在影响

- **资金被完全抽干：**攻击者可以利用该漏洞重复提取资金，直至智能合约余额耗尽。
- **合约状态被篡改：**攻击者可以在状态未更新的情况下，执行未授权操作，导致合同或相关系统内执行非预期的操作。

修复方案

- 始终确保在调用外部合约之前完成所有状态变化，即在调用外部代码之前更新内部余额或代码。
- 使用防止重入的函数修饰符，例如 Open Zeppelin 的重入防护（Re-entrancy Guard）。

真实案例

1. Rari Capital : A Comprehensive Hack Analysis
2. Orion Protocol : A Comprehensive Hack Analysis

示例 1_存在漏洞的合约

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_Reentrancy {
    mapping(address => uint) public balances;

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

    function withdraw() external {
        uint amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");
```

```
        // Vulnerability: Ether is sent before updating the user's balance, allowing reentrancy.（漏洞：在更新用户余额之前先转账，导致可重入攻击）
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");

        // Update balance after sending Ether（余额在资金转出后才更新，攻击者可以在此之前反复调用 withdraw）
        balances[msg.sender] = 0;
    }
}
```

示例 2_修复后的合约

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_Reentrancy {
    mapping(address => uint) public balances;

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

    function withdraw() external {
        uint amount = balances[msg.sender];
        require(amount > 0, "Insufficient balance");

        // Fix: Update the user's balance before sending Ether（先更新余额，避免可重入攻击）
        balances[msg.sender] = 0;

        // Then send Ether（再执行转账）
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

SC06:2025 未检查的外部调用

描述

未检查的外部调用是一种常见的智能合约安全漏洞，发生在合约调用外部合约或地址时未正确验证调用结果。在以太坊中，如果一个合约调用另一个合约，该调用可能悄无声息地失败，而不会抛出异常。如果调用合约未检查返回值，则可能错误地认为调用成功，从而导致状态不一致或漏洞被攻击者利用。

潜在影响

- **交易失败但未被检测：**合约可能继续执行后续逻辑，而实际上操作未成功，导致状态异常。
- **资金损失：**如果调用失败但未检查返回值，可能导致合约误判资金已转移，最终造成资产损失。
- **逻辑不一致：**由于未检测外部调用是否成功，可能导致合约数据不准确，进一步影响其安全性和可预测性。

修复方案

- **使用 transfer() 代替 send()：**transfer() 在调用失败时会自动回滚交易，减少错误执行的风险。
- **始终检查 send() 或 call() 返回值：**确保外部调用成功后才执行后续逻辑，否则应中止执行。
- 在 delegatecall 之后检查返回状态，避免未成功执行代码导致逻辑漏洞。（译者补充）

真实案例

1. Punk Protocol Hack : A Comprehensive Hack Analysis

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.4.24;

contract Solidity_UncheckedExternalCall {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    function forward(address callee, bytes _data) public {
        require(callee.delegatecall(_data));  // 未检查返回值，可能导致错误执行）
```

```
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_UncheckedExternalCall {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function forward(address callee, bytes memory _data) public {
        // Ensure that delegatecall succeeds（确保 delegatecall 成功）
        (bool success, ) = callee.delegatecall(_data);
        require(success, "Delegatecall failed");  // Check the return value to handle failure（检查返回值，确保外部调用正确执行）
    }
}
```

SC07:2025 闪电贷攻击

描述

闪电贷攻击利用了智能合约的一种特殊机制，即无需抵押即可在单笔交易中借入巨额资金。由于区块链交易的原子性（所有操作要么全部成功，要么全部失败），攻击者可以在偿还贷款前操纵市场价格、触发清算、或利用其他漏洞（如预言机操纵、可重入攻击或逻辑错误）破坏智能合约的正常运作，从而获取巨额利润。

闪电贷攻击常见手法

- **预言机操纵**：使用闪电贷借来的资金短时间内**人为干预价格预言机**，导致合约计算错误，引发抵押不足的清算。
- **流动性池枯竭**：利用闪电贷在自动做市商（AMM）中操作价格，导致资金池资产失衡或被耗尽。
- **跨平台套利**：在多个交易平台之间利用价格差异套利，放大市场波动并获取不当收益。

潜在影响

- **资金损失**：攻击者可以迅速抽空协议储备资金，或利用抵押贷款机制窃取资产。
- **市场波动**：短时间的价格操纵或流动性枯竭可能影响用户和平台。
- **生态系统信任受损**：用户对受攻击的协议的信任度下降，从而导致资金流失和用户减少。

修复方案

- **避免依赖闪电贷执行关键业务逻辑**：关键合约功能应在可验证和可预测的条件下运行，防止被闪电贷操控。
- **加强预言机设计**：采用时间加权平均价格（TWAP）或去中心化预言机，提高价格数据的稳定性，减少短时间内的价格操纵风险。
- **全面测试与模拟攻击**：在安全审计和合约测试中模拟闪电贷攻击场景，并边界条件的测试，确保合约能够抵御极端情况。
- **访问控制**：限制关键函数的访问权限，以防止未经授权或恶意交易者利用闪电贷机制操纵合约。

真实案例

1. UwUlend Hack: A Comprehensive Hack Analysis
2. Doughfina Hack: A Comprehensive Hack Analysis

SC08:2025 整数溢出与下溢

描述

以太坊虚拟机（EVM）中的整数是固定大小的数据类型，这意味着每种整数类型的值域是有限的。例如，uint8（8 位无符号整数）只能存储 0 到 255 之间的数值。如果尝试存储超过 255 的数值，就会发生溢出（Overflow），而如果从 0 减去 1，就会发生下溢（Underflow），最终数值会绕回 255。

在有符号整数（如 int8）中，行为略有不同。例如，int8 类型的最小值是 -128，如果从 -128 减去 1，则结果会变为 127（绕回最大值）。这种算术运算超出数据类型范围的情况，可能会导致智能合约逻辑被绕过，甚至被攻击者恶意利用。

Solidity 0.8.0 及以上版本的改进

Solidity 0.8.0 及以上版本自动检测整数溢出和下溢，如果发生超出范围的计算，交易会自动回滚。此外，Solidity 0.8.0 引入了 unchecked 关键字，允许开发者显式绕过这些检查，以优化 Gas 费用，适用于某些情况下溢出行为是可接受的场景。

潜在影响

- **操纵账户余额或代币数量：**攻击者可能通过溢出漏洞人为增加账户余额，获取超出其实际拥有的资产。
- **破坏合约逻辑：**攻击者可以利用溢出漏洞绕过合同逻辑的预期流程，导致如资产盗窃或过量铸造代币等未经授权的行为。

修复方案

- **使用 Solidity 0.8.0 及以上版本：**新版编译器默认检测溢出和下溢，确保算术运算安全。
- **使用最新的安全数学库（SafeMath）：**对于以太坊社区来说，OpenZeppelin 在创建和审核安全库方面做得非常出色。尤其是其 SafeMath 库，可用于防止欠载/溢出漏洞。它提供了 add()、sub()、mul() 等函数，可执行基本算术运算，并在发生溢出或下溢时自动恢复。

真实案例

1. PoWH Coin Ponzi Scheme : A Comprehensive Hack Analysis
2. Poolz Finance : A Comprehensive Hack Analysis

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.4.17;
```

```
contract Solidity_OverflowUnderflow {
    uint8 public balance;

    constructor() public {
        balance = 255; // Maximum value of uint8（uint8 的最大值）
    }

    // Increments the balance by a given value（增加余额）
    function increment(uint8 value) public {
        balance += value; // Vulnerable to overflow（可能导致溢出）
    }

    // Decrements the balance by a given value（减少余额）
    function decrement(uint8 value) public {
        balance -= value; // Vulnerable to underflow（可能导致下溢）
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Solidity_OverflowUnderflow {
    uint8 public balance;

    constructor() {
        balance = 255; // Maximum value of uint8（ uint8 的最大值）
    }

    // Increments the balance by a given value（增加余额）
    function increment(uint8 value) public {
        balance += value; // Solidity 0.8.x automatically checks for overflow（Solidity 0.8.x 自动检测溢出）
    }
```

```
    //Decrements the balance by a given value（ 减少余额，确保不会发生下溢）
    function decrement(uint8 value) public {
        require(balance >= value, "Underflow detected");
        balance -= value;
    }
}
```

SC09:2025 不安全的随机数

描述

随机数生成在博彩、游戏赢家选择、随机种子生成等应用场景中至关重要。然而，由于 Ethereum 是一个确定性系统，在 Solidity 中生成真正的随机数极具挑战性。Solidity 无法直接生成真正的随机数，而是依赖于伪随机数，但这些方法极易被攻击者预测或操控。此外，复杂的随机数计算也会消耗大量 Gas，影响合约的可扩展性和成本。

常见的不安全随机数生成方式：开发者通常会使用以下区块属性作为随机数的输入来源，

- block.timestamp：当前区块的时间戳
- blockhash(uint blockNumber)：指定区块的哈希值（仅限最近 256 个区块）
- block.difficulty：当前区块的挖矿难度
- block.number：当前区块号
- block.coinbase：当前区块的矿工地址

这些方法存在安全风险，因为矿工可以操控区块属性，从而影响智能合约的随机数生成逻辑，导致攻击者能够预测或控制随机结果。

潜在影响

- **攻击者可预测随机数：**利用可控的区块属性，攻击者可以计算出“随机”值，确保自己获胜，例如操控 博彩、抽奖、游戏奖励 机制。
- **影响公平性：**如果随机数生成方式可被操控，将导致智能合约的公平性和完整性受到严重威胁，损害用户信任。
- **经济损失：**由于随机结果可被预测，恶意用户可以利用此漏洞获得超额奖励，破坏项目的经济模型。

修复方案

- **使用预言机：**使用语言机（如 Oraclize）作为外部随机性来源。在信任预言机时需谨慎。也可以同时使用多个预言机。
- **采用承诺-揭示机制：**具有广泛的应用，如硬币抛掷、零知识证明和安全计算。例如：RANDAO。
- **Chainlink VRF：**一个可证明公平且可验证的随机数生成器（RNG），使智能合约能够在不牺牲安全性或可用性的情况下访问随机值。
- **Signidice 算法：**适用于双方使用秘密技术的应用程序中的伪随机数生成（PRNG）。
- **引入比特币区块哈希：**通过比特币区块链获取未来区块哈希值，作为以太坊合约的随机数来源（需谨慎使用，可能存在矿工激励攻击）。

真实案例

1. Roast Football Hack : A Comprehensive Hack Analysis
2. FFIST Hack : A Comprehensive Hack Analysis

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract Solidity_InsecureRandomness {
    constructor() payable {}

    function guess(uint256 _guess) public {
        uint256 answer = uint256(
            keccak256(
                abi.encodePacked(block.timestamp, block.difficulty, msg.sender) //Using insecure mechanisms for random number generation（ 使用不安全的随机数生成方式）
            )
        );

        if (_guess == answer) {
            (bool sent,) = msg.sender.call{value: 1 ether}("");
            require(sent, "Failed to send Ether");
        }
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import "@chainlink/contracts/src/v0.8/VRFConsumerBase.sol";

contract Solidity_InsecureRandomness is VRFConsumerBase {
    bytes32 internal keyHash;
    uint256 internal fee;
```

```
    uint256 public randomResult;

    constructor(address _vrfCoordinator, address _linkToken, bytes32 _keyHash, uint256 _fee)
        VRFConsumerBase(_vrfCoordinator, _linkToken)
    {
        keyHash = _keyHash;
        fee = _fee;
    }

    function requestRandomNumber() public returns (bytes32 requestId) {
        require(LINK.balanceOf(address(this)) >= fee, "Not enough LINK");
        return requestRandomness(keyHash, fee);
    }

    function fulfillRandomness(bytes32 requestId, uint256 randomness) internal override {
        randomResult = randomness;
    }

    function guess(uint256 _guess) public {
        require(randomResult > 0, "Random number not generated yet");
        if (_guess == randomResult) {
            (bool sent,) = msg.sender.call{value: 1 ether}("");
            require(sent, "Failed to send Ether");
        }
    }
}
```

SC10:2025 拒绝服务（DoS）攻击

描述

- Solidity 中的拒绝服务（DoS）攻击涉及利用漏洞消耗 GAS、CPU 周期或存储等资源，使智能合约无法使用。常见的类型包括 GAS 耗尽攻击（恶意行为者创建需要过多 GAS 的交易）、重入性攻击（利用合约调用序列访问未经授权的资金）和区块 GAS 限制攻击（消耗区块 GAS，阻碍合法交易）。

潜在影响

- **智能合约陷入不可用状态：**攻击者可阻止合法用户交互，影响核心业务运作。
- **导致资金损失：**如果合约涉及资金管理或资产的去中心化应用（dApps）中，DoS 攻击可能导致提款受阻或资产被锁定。
- **损害平台信誉：**用户可能对遭受 DoS 攻击的平台的安全性和可靠性失去信任，从而导致用户流失和业务机会减少。

修复方案

- **确保智能合约能够处理一致的故障，例如对可能失败的外部调用进行异步处理，以维护合约的完整性并防止意外行为。**
- **在使用调用进行外部调用、循环和遍历时要小心，以避免过多的 GAS 消耗，这可能导致交易失败或意外成本。**
- **避免在合约权限中过度授权单一角色。相反，应合理分配权限，并对具有关键权限的角色使用多重签名钱包管理，以防止因私钥泄露而导致的权限丢失。**

示例 1（存在漏洞的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract Solidity_DOS {
    address public king;
    uint256 public balance;

    function claimThrone() external payable {
        require(msg.value > balance, "Need to pay more to become the king");

        // If the current king has a malicious fallback function that reverts, it will prevent the new king from claiming the throne, causing a Denial of Service.（如果当前国王的 fallback 函数恶意回滚，则新国王无法成功登基，导致 DoS 攻击）
        (bool sent,) = king.call{value: balance}("");
```

```
        require(sent, "Failed to send Ether");

        balance = msg.value;
        king = msg.sender;
    }
}
```

示例 2（修复后的合约）

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract Solidity_DOS {
    address public king;
    uint256 public balance;

    // Use a safer approach to transfer funds, like transfer, which has a fixed gas stipend.（采用更安全的转账方式，如 transfer，它有固定 Gas 限制。）
    // This avoids using call and prevents issues with malicious fallback functions.（避免 malicious fallback 影响交易。）
    function claimThrone() external payable {
        require(msg.value > balance, "Need to pay more to become the king");

        address previousKing = king;
        uint256 previousBalance = balance;

        // Update the state before transferring Ether to prevent reentrancy issues.（先更新状态，防止可重入攻击）
        king = msg.sender;
        balance = msg.value;

        // Use transfer instead of call to ensure the transaction doesn't fail due to a malicious fallback.（使用 transfer 而非 call，确保交易不会因恶意 fallback 失败）
        payable(previousKing).transfer(previousBalance);
    }
}
```