“敬畏之心”，Web3的安全性

智能合约安全 (-) 控制权与闪电贷攻击

这个智能合约安全系列提供了一个广泛的列表，列出了在 Solidity 智能合约中容易反复出现的问题和漏洞。

Solidity 中的安全问题可以归结为智能合约的行为方式不符合它们的意图。这可以分为四大类：

- 资金被盗
- 资金被锁住或冻结在合约内
- 人们收到的奖励比预期的少（奖励被延迟或减少）。
- 人们收到的奖励比预期的多（导致通货膨胀和贬值）。

我们不可能对所有可能出错的事情做一个全面的列表。然而，正如传统的软件工程有常见的漏洞主题，如 SQL 注入、缓冲区超限和跨网站脚本，智能合约中也有反复出现的反模式(anti-pattern)。

智能合约的黑客和漏洞

把这个系列看作是一本书，不反复推敲，就不可能详细讨论每一个概念。

然而，这个系列可以作为一个清单，说明应该注意什么和研究什么。如果一个主题感觉不熟悉，这应该作为一个指标，说明值得花时间去练习识别该类漏洞。

前提条件

假设你对 Solidity 有基本的熟练程度。如果你是 Solidity 的新手，请看我们的 免费 Solidity 教程。

重入攻击

每当一个智能合约调用另一个智能合约的函数，向其发送以太币，或向其代币转账，那么就有可能出现重入。

当以太币被转账时，接收合约的回退或接收函数被调用。这就把控制权交给了接收方。

一些代币协议通过调用一个预先确定的函数来提醒接收智能合约收到了代币。这就把控制流交给了接收函数。

当攻击合约收到控制权时，它不一定要调用交出控制权的同一个函数。它可以调用受害者智能合约中的不同函数（跨函数重入），甚至是不同的合约（跨合约重入）。

只读重入发生在合约处于中间状态时访问一个视图函数。

尽管重入可能是最知名的智能合约漏洞，但它只占发生的黑客的一小部分。安全研究员 Pascal Caversaccio(pcaveraccio)在 github 上保持着一个最新的重入攻击列表。截至 2023 年 4 月，该库中已经记录了 46 个重入攻击。

访问控制

这似乎是一个简单的错误，但忘记对谁可以调用一个敏感函数（如提取以太币或改变所有权）进行限制，这种情况经常发生，令人惊讶。

即使修改器已经写了，也会修改器没有正确实现的情况，比如下面的例子中缺少 require 语句：

// DO NOT USE!

modifier onlyMinter {

minters[msg.sender] == true_;

}

上面这个代码是这次审计中的一个真实例子：https://code4rena.com/reports/2023-01-rabbithole/#h-01-bad-implementation-in-minter-access-control-for-rabbitholereceipt-and-rabbitholetickets-contracts

这里是访问控制出错的另一种方式：

```
    function claimAirdrop(bytes32 calldata proof[]) {

        bool verified = MerkleProof.verifyCalldata(proof, merkleRoot,
  keccak256(abi.encode(msg.sender)));
        require(verified, "not verified");
        require(alreadyClaimed[msg.sender], "already claimed");

        _transfer(msg.sender, AIRDROP_AMOUNT);

    }
```

在此案例中，"alreadyClaimed "永远不会被设置为真，所以申领者可以发出多次调用该函数。

现实案例：交易机器人被利用

最近的一个访问控制不足的例子是，一个交易机器人（它的名字叫 0xbad，因为地址是以其开头）接收闪电贷的函数没有受到保护。它积累了超过一百万美元的利润，直到有一天，攻击者注意到任何地址都可以调用 flashloan 接收函数，而不仅仅是 flashloan 提供者。

正如交易机器人通常的情况一样，执行交易的智能合约代码没有经过验证，但攻击者还是发现了这个弱点。更多信息见 rekt news 报道。

不正确的输入验证

如果访问控制是关于控制谁调用一个函数，那么输入验证就是控制他们用什么来调用合约。

这通常归结为忘记在适当的地方设置 require 语句。

下面是一个基本的例子：

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

```

```
    // allow depositing on other's behalf
    function deposit(address for) public payable {
        balances += msg.value;
    }

    function withdraw(address from, uint256 amount) public {
        require(balances[from] <= amount, "insufficient balance");

        balances[from] -= amount;
        msg.sender.call{value: amout}("");
    }

}
```

上面的合约确实检查了你提取的金额没有超过你账户中的金额，但它并没有阻止你从一个任意的账户中提取。

现实案例：Sushiswap

Sushiswap 由于一个外部函数的一个参数没有被检查，经历了一次这种类型的黑客攻击。

不适当的访问控制和不适当的输入验证之间有什么区别?

不适当的访问控制意味着 msg.sender 没有足够的限制。不当的输入验证意味着函数的参数没有得到充分的处理。这种反模式还有一个反面：在函数调用中放置过多的限制。

过多的函数限制

过多的验证可能意味着资金不会被盗，但它可能意味着资金被锁定在合约中。拥有多重的保障措施也不是一件好事。

现实生活中的例子：Akutars NFT

最引人注目的事件之一是 Akutars NFT，它最终导致价值 3400 万美元的 Eth 卡在智能合约内，无法提取。

该合约有一个用心良苦的机制，以防止合约的所有者退出，直到所有付款超过荷兰拍卖价格的退款被给予。但由于下面链接的 Twitter 线程中记录的一个错误，所有者无法提取资金。

掌握好平衡

Sushiswap 给了不被信任的用户太多的权力，而 Akutars NFT 给管理员的权力太少。在设计智能合约时，必须对每一类用户的自由度做出主观判断，这个决定不能留给自动化测试和工具。必须考虑与去中心化、安全和用户体验的重大权衡。

对于智能合约程序员来说，明确写出用户应该和不应该对某些函数做什么是开发过程中的一个重要部分。

我们将在后面重新审视权力过大的管理员这个话题。

安全问题可以归结为管理资金退出合约的方式

正如介绍中所说，智能合约被黑的主要方式有四种：

金钱被盗

钱被冻结

奖励不足

奖励过高

这里的 "钱 "是指任何有价值的东西，如代币，而不仅仅是加密货币。在对智能合约进行编码或审计时，开发人员必须有意识地了解价值流入和流出合约的预期方式。上面列出的问题是智能合约被黑的主要途径，但还有很多其他的根本原因，可以串联成重大问题，下面记录了这些问题。

双重投票或 msg.sender 欺骗

使用 vanilla ERC20 代币或 NFT 作为门票来计算投票权重是不安全的，因为攻击者可以用一个地址投票，将代币转账到另一个地址，并从该地址再次投票。

下面是一个最小案例：

```
// A malicious voter can simply transfer their tokens to
// another address and vote again.
contract UnsafeBallot {

    uint256 public proposal1VoteCount;
    uint256 public proposal2VoteCount;

    IERC20 immutable private governanceToken;

    constructor(IERC20 _governanceToken) {
        governanceToken = _governanceToken;
    }

    function voteFor1() external notAlreadyVoted {
        proposal1VoteCount += governanceToken.balanceOf(msg.sender);
    }

    function voteFor2() external notAlreadyVoted {
        proposal2VoteCount += governanceToken.balanceOf(msg.sender);
    }

    // prevent the same address from voting twice,
    // however the attacker can simply
    // transfer to a new address
    modifier notAlreadyVoted {
        require(!alreadyVoted[msg.sender], "already voted");
        _;
        alreadyVoted[msg.sender] = true;
    }

}
```

为了防止这种攻击，应该使用 ERC20 Snapshot 或 ERC20 Votes。通过对过去的一个时间点进行快照，当前的代币余额不能被操纵以获得非法投票权。

闪电贷治理攻击

然而，使用具有快照或投票函数的 ERC20 代币并不能完全解决这个问题，如果有人可以通过闪电贷来暂时增加他们的余额，然后在同一交易中对他们的余额进行快照。如果该快照被用于投票，他们将有一个不合理的大量投票权可供支配。

闪存贷款将大量的以太币或代币借给一个地址，但如果这笔钱没有在同一次交易中偿还，就会回退。

```
contract SimpleFlashloan {

    function borrowERC20Tokens() public {
        uint256 before = token.balanceOf(address(this));

        // send tokens to the borrower
        token.transfer(msg.sender, amount);

        // hand control back to the borrower to
        // let them do something
        IBorrower(msg.sender).onFlashLoan();

        // require that the tokens got returned
        require(token.balanceOf(address(this) >= before);
    }

}
```

攻击者可以利用 flashloan 突然获得大量的选票，使提案对他们有利和/或做一些恶意的事情。

闪电贷价格攻击

这可以说是对 DeFi 最常见的（或至少是最引人注目的）攻击，占了数亿美元的损失。这里有一个列表。

区块链上的资产价格通常被计算为资产之间的当前汇率。例如，如果一个合约目前是 1 美元兑 100 个 k9 币，那么你可以说 k9 币的价格是 0.01 美元。然而，价格通常会随着买卖压力的变化而变化，而闪电贷会产生巨大的买卖压力。

当查询另一个智能合约的资产价格时，开发者需要非常小心，因为他们假设他们所调用的智能合约对闪电贷的操纵是免疫的。

绕过合约检查

你可以通过查看一个地址的字节码大小来 "检查 "它是否是一个智能合约。外部拥有的账户（普通钱包）没有任何字节码。这里有几种方法可以做到这一点

```
import "@openzeppelin/contracts/utils/Address.sol"
contract CheckIfContract {

    using Address for address;

    function addressIsContractV1(address _a) {
        return _a.code.length == 0;
    }

    function addressIsContractV2(address _a) {

        // use the openzeppelin libraryreturn _a.isContract();
    }

}
```

然而，这有一些限制

如果一个合约从构造函数中进行外部调用，那么它的表面字节码大小将为零，因为智能合约部署代码还没有返回运行时代码

该代码现在可能是空的，但攻击者可能知道他们可以在未来使用 create2 在那里部署一个智能合约。

一般来说，检查一个地址是否是合约通常是（但不总是）一种反模式。多签名钱包本身就是智能合约，做任何可能破坏多签名钱包的事情都会破坏可组合性。

这方面的例外是在调用转移钩子之前检查目标是否是智能合约。稍后会有更多关于这方面的内容。

tx.origin

使用 tx.origin 很少有好的理由。如果 tx.origin 被用来识别交易发起人，那么中间人攻击是可能的。如果用户被骗去调用一个恶意的智能合约，那么该智能合约就可以利用 tx.origin 所拥有的所有权限来进行破坏。

请考虑下面这个练习，以及代码上方的注释。

```
contract Phish {
function phishingFunction() public {

        // this fails, because this contract does not have approval/allowance
        token.transferFrom(msg.sender, address(this),
token.balanceOf(msg.sender));

        // this also fails, because this creates approval for the contract,//
not the wallet calling this phishing function
        token.approve(address(this), type(uint256).max);
    }

}
```

这并不意味着你可以安全地调用任意的智能合约。但大多数协议中都有一层安全机制，如果使用 tx.origin 进行认证，就会被绕过。

有时，你可能会看到像这样的代码：

require(msg.sender == tx.origin, "no contracts");

当一个智能合约调用另一个智能合约时，msg.sender 将是智能合约，tx.origin 将是用户的钱包，从而给出一个可靠的指示，即传入的调用是来自智能合约。即使调用发生在构造函数中也是如此。

大多数情况下，这种设计模式不是一个好主意。多签名钱包和来自 EIP 4337 的钱包将无法与具有这种代码的函数交互。这种模式通常可以在 NFT mint 中看到，在那里，我们有理由相信大多数用户都在使用传统的钱包。但随着账户抽象化的普及，这种模式的阻碍作用将大于其帮助。

Gas 导致拒绝服务

悲痛攻击（griefing attack）意味着黑客试图为其他人 "制造悲痛"，即使他们没有从这样做中获得经济利益。

一个智能合约可以通过进入一个无限循环，恶意地用完转发给它的所有 Gas。考虑一下下面的例子：

```
contract Mal {

    fallback() external payable {

        // infinite loop uses up all the gas
        while (true) {

        }
    }

}
```

如果另一个合约将以太坊分配到一个地址列表中，如以下：

```
contract Distribute {
funtion distribute(uint256 total) public nonReentrant {
for (uint i; i < addresses.length; ) {

            (bool ok, ) addresses.call{value: total / addresses.length}("");
            // ignore ok, if it reverts we move on
            // traditional gas saving trick for for loops
            unchecked {
                ++i;
            }
        }
    }

}
```

那么该函数将在向 Mal 合约发送以太币时进行还原。上面代码中的调用转发了 63 / 64 的可用 Gas，所以很可能没有足够的 Gas 来完成只剩下 1/64 的 Gas 的操作。

一个智能合约可以返回一个消耗大量 Gas 的大型内存数组

考虑下面的例子

```
function largeReturn() public {

    // result might be extremely long!
    (book ok, bytes memory result) =
        otherContract.call(abi.encodeWithSignature("foo()"));

    require(ok, "call failed");

}
```

内存数组在 724 字节之后使用了四次方的 Gas，所以仔细选择的返回数据大小可以让调用者感到悲伤。

即使变量的结果没有被使用，它仍然被复制到内存中。如果你想把返回的大小限制在一定范围内，你可以使用汇编

```
function largeReturn() public {
assembly {
let ok := call(gas(), destinationAddress, value, dataOffset, dataSize, 0x00,
0x00);
// nothing is copied to memory until you
// use returndatacopy()
}
}
```

删除别人可以添加的数组也是一个拒绝服务的向量

尽管擦除存储是一个节省 Gas 的操作，但它仍然有一个成本。如果一个数组变得太长，它就不可能被删除。下面是一个最小案例

```
contract VulnerableArray {

    address[] public stuff;

    function addSomething(address something) public {
        stuff.push(something);
    }

    // 如果 stuff 太长,由于 gas 问题将无法删除
    function deleteEverything() public onlyOwner {
        delete stuff;
    }

}
```

ERC777, ERC721, 和 ERC1155 也可以是悲痛的向量

如果一个智能合约转账到有转账 hook 的代币，攻击者可以设置一个不接受代币的合约（它要么没有 onReceive 函数，要么将该函数编程为回退）。这将使代币无法转账，并导致整个交易被回退。

在使用 safeTransfer 或 transfer 之前，要考虑到接收方可能会强迫交易回退的可能性。

```
contract Mal is IERC721Receiver, IERC1155Receiver, IERC777Receiver {

    // this will intercept any transfer hook
    fallback() external payable {

        // infinite loop uses up all the gaswhile (true) {
```

```
    }
}

    // we could also selectively deny transactions
    function onERC721Received(address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external returns (bytes4) {

        if (wakeUpChooseViolence()) {
            revert();
        }
        else {
            return IERC721Receiver.onERC721Received.selector;
        }
    }

}
```

智能合约安全 (二) 不安全的随机数&预言机

目前不可能用区块链上的单一交易安全地产生随机数。区块链需要是完全确定的，否则分布式节点将无法达成关于状态的共识。因为它们是完全确定的，所以任何 "随机"的数字都可以被预测到。下面的掷骰子函数可以被利用。

```
contract UnsafeDice {
    function randomness() internal returns (uint256) {
        return keccak256(abi.encode(msg.sender, tx.origin, block.timestamp, tx.gasprice, blockhash(block.number - 1));
    }

    // our dice can land on one of {0,1,2,3,4,5}function rollDice() public payable {
        require(msg.value == 1 ether);

        if (randomness() % 6) == 5) {
            msg.sender.call{value: 2 ether}("");
```

```
        }
    }
}

contract ExploitDice {
    function randomness() internal returns (uint256) {
        return keccak256(abi.encode(msg.sender, tx.origin, block.timestamp, tx.gasprice, blockhash(block.number - 1));
    }

    function betSafely(IUnsafeDice game) public payable {
        if (randomness % 6) == 5)) {
            game.betSafely{value: 1 ether}()
        }

        // else don't do anything
    }
}
```

如何来产生随机数并不重要，因为攻击者可以完全复制它。使用更多的 "熵"的来源，如 msg.sender、时间戳等，不会有任何影响，因为智能合约也可以预测它。

错误使用 Chainlink 随机数 Oracle

Chainlink 是一个流行的解决方案，以获得安全的随机数。它分两步进行。首先，智能合约向预言机处发送一个随机数请求，然后在一些区块之后，预言机以一个随机数作为回应。

由于攻击者无法预测未来，所以他们无法预测随机数。

除非智能合约错误地使用预言机：

请求随机数的智能合约必须在随机数返回之前不做任何事情。否则，攻击者可以监视返回随机数的预言机的 mempool，并在前面运行预言机，知道随机数会是什么。

随机数预言机本身可能会试图操纵你的应用程序。如果没有其他节点的共识，他们不能挑选随机数，但如果你的应用程序同时请求几个随机数，他们可以扣留和重新排序。

最终性在以太坊或大多数其他 EVM 链上不是即时的。仅仅因为某些区块是最新的，并不意味着它不一定会保持这种状态。这被称为 "链上重组"。事实上，链可以改变的不仅仅是最后一个区块。这就是所谓的 "深度重组"。Etherscan 报告了各种链的 re-orgs，例如以太坊重组和 Polygon 重组。在 Polygon 上，重组的深度可以达到 30 个或更多的区块，所以等待更少的区块会使应用变得脆弱（当 zk-evm 成为 Polygon 上的标准共识时，这种情况可能会改变，因为最终性将与以太坊的一致，但这是未来的预测，而不是目前的事实）。

下面是其他 Chainlink 随机数的安全考虑。

从价格 Oracle 中获取陈旧的数据

Chainlink 没有 SLA（服务水平协议）来保持它的价格预言机在一定时间范围内的更新。当链上的交易严重拥堵时，价格更新可能会被延迟。

使用价格预言机的智能合约必须明确地检查数据是否陈旧，即最近在某个阈值内被更新。否则，它不能对价格做出可靠的决策。

还有一个更复杂的问题，如果价格没有变化超过偏差阈值，预言机可能不会更新价格以节省 Gas，所以这可能会影响到什么时间阈值被认为是 "陈旧"。

了解智能合约所依赖的 Oracle 的服务水平协议是很重要的。

只依赖一个预言机

无论一个预言机看起来多么安全，将来都可能发现攻击。对此的唯一防御措施是使用多个独立的预言机。

一般来说，预言机是很难正确的

区块链可以是相当安全的，但首先把数据放到链上就必须进行某种链外操作，这就放弃了区块链提供的所有安全保证。即使预言机者保持诚实，他们的数据来源也可以被操纵。例如，一个信使可以可靠地报告来自中心化交易所的价格，但这些价格可以被大量的买入和卖出订单所操纵。同样，依赖于传感器数据或一些 web2 API 的预言机也会受到传统黑客攻击的影响。

一个好的智能合约架构在可能的情况下会完全避免使用预言机。

混合计算

考虑以下合约

```
contract MixedAccounting {
    uint256 myBalance;

    function deposit() public payable {
        myBalance = myBalance + msg.value;
    }

    function myBalanceIntrospect() public view returns (uint256) {
        return address(this).balance;
    }

    function myBalanceVariable() public view returns (uint256) {
        return myBalance;
    }

    function notAlwaysTrue() public view returns (bool) {
        return myBalanceIntrospect() == myBalanceVariable();
    }
}
```

上面的合约没有接收或回退函数，所以直接将以太传送给它就会回退。然而，合约可以用自毁的方式强行向它发送以太。在此案例中，myBalanceIntrospect()会比 myBalanceVariable() 大。两种以太币的计算方法都没有问题，但如果你同时使用这两种方法，那么合约可能会有不一致的行为。

这同样适用于 ERC20 代币。

```
contract MixedAccountingERC20 {

```

```
    IERC20 token;
    uint256 myTokenBalance;

    function deposit(uint256 amount) public {
        token.transferFrom(msg.sender, address(this), amount);
        myTokenBalance = myTokenBalance + amount;
    }

    function myBalanceIntrospect() public view returns (uint256) {
        return token.balanceOf(address(this));
    }

    function myBalanceVariable() public view returns (uint256) {
        return myTokenBalance;
    }

    function notAlwaysTrue() public view returns (bool) {
        return myBalanceIntrospect() == myBalanceVariable();
    }
}
```

我们再次不能假设 myBalanceIntrospect()和 myBalanceVariable()总是返回相同的值。可以直接将 ERC20 代币转账到 MixedAccountingERC20，绕过存款函数，不更新 myTokenBalance 变量。

在用反省检查余额时，应避免严格使用相等检查，因为余额可以被外人随意改变。

把加密证明当作密码一样对待

这不是 Solidity 的一个怪癖，更多的是开发者对如何使用密码学来赋予地址特殊权限有普遍误解。下面的代码是不安全的：

```
contract InsecureMerkleRoot {
    bytes32 merkleRoot;
    function airdrop(bytes[] calldata proof, bytes32 leaf) external {

        require(MerkleProof.verifyCalldata(proof, merkleRoot, leaf), "not verified");
        require(!alreadyClaimed[leaf], "already claimed airdrop");
```

```
        alreadyClaimed[leaf] = true;

        mint(msg.sender, AIRDROP_AMOUNT);
    }
}
```

这段代码是不安全的，原因有三：

任何知道被选中进行空投的地址的人都可以重新创建 Merkle 树并创造一个有效的证明。

叶子没有 Hash。攻击者可以提交一个与 Merkle 根相同的叶子，并绕过 require 语句。

即使上述两个问题被修复，一旦有人提交了有效的证明，他们就可以被抢跑。

加密证明（Merkle 树、签名等）需要与 msg.sender 绑定，攻击者在没有获得私钥的情况下无法操纵。

智能合约安全 (三) 不安全的代币

ERC20 代币问题

如果你只处理受信任的 ERC20 代币，这些问题大多不适用。然而，当与任意的或部分不受信任的 ERC20 代币交互时，就有一些需要注意的地方。

ERC20：转账扣费

当与不信任的代币打交道时，你不应该认为你的余额一定会增加那么多。一个 ERC20 代币有可能这样实现它的转账函数，如下所示：

```
contract ERC20 {

    // internally called by transfer() and transferFrom()
    // balance and approval checks happen in the caller
```

```
    function _transfer(address from, address to, uint256 amount) internal
returns (bool) {
        fee = amount * 100 / 99;

        balanceOf[from] -= to;
        balanceOf[to] += (amount - fee);

        balanceOf[TREASURY] += fee;

        emit Transfer(msg.sender, to, (amount - fee));
        return true;
    }
}
```

这种代币对每笔交易都会征收 1%的税。因此，如果一个智能合约与该代币进行如下交互，我们将得到意想不到的回退或资产被盗。

```
contract Stake {

    mapping(address => uint256) public balancesInContract;

    function stake(uint256 amount) public {
        token.transferFrom(msg.sender, address(this), amount);
        balancesInContract[msg.sender] += amount; //  这是错误的
    }

    function unstake() public {
        uint256 toSend = balancesInContract[msg.sender];
        delete balancesInContract[msg.sender];

        // this could revert because toSend is 1% greater than
        // the amount in the contract. Otherwise, 1% will be "stolen"// from
other depositors.
        token.transfer(msg.sender, toSend);
    }
}
```

ERC20: rebase 的代币

Rebasing 代币由 Olympus DAO 的 sOhm 代币 和 Ampleforth 的 AMPL 代币所推广。Coingecko 维护了一个 Rebasing ERC20 代币的列表。

当一个代币回溯时，总发行量会发生变化，每个人的余额会根据回溯的方向而增加或减少。

在处理 rebase 代币时，以下代码可能会被破坏：

```
contract WillBreak {
    mapping(address => uint256) public balanceHeld;
    IERC20 private rebasingToken

    function deposit(uint256 amount) external {
        balanceHeld[msg.sender] = amount;
        rebasingToken.transferFrom(msg.sender, address(this), amount);
    }

    function withdraw() external {
        amount = balanceHeld[msg.sender];
        delete balanceHeld[msg.sender];

        // 错误，amount 也许会超出转出范围
        rebasingToken.transfer(msg.sender, amount);
    }
}
```

许多合约的解决方案是简单地不允许 rebase 代币。然而，我们可以修改上面的代码，在将账户余额转给接受者之前检查 balanceOf(address(this))。那么，即使余额发生变化，它仍然可以工作。

ERC20: ERC777 在 ERC20 上的包裹

ERC20，如果按照标准实现，ERC20 代币没有转账钩子（hook），因此 transfer 和 transferFrom 不会有重入问题。

带有转账钩子的代币有应用优势，这就是为什么所有的 NFT 标准都实现了它们，以及为什么 ERC777 被最终确定。然而，这已经引起了足够的混乱，以至于 Openzeppelin 废止了 ERC777 库。

如果你只想让你的协议与那些行为像 ERC20 代币但有转账 hook 的代币兼容，那么这只是一个简单的问题，把 transfer 和 transferFrom 函数当作它们会向接收者进行一个函数调用即可。

这种 ERC777 的重入发生在 Uniswap 身上（如果你好奇，Openzeppelin 在这里记录了这个漏洞）。

ERC20: 不是所有的 ERC20 代币转账都会返回 true

ERC20 规范规定，ERC20 代币在转账成功时必须返回 true。因为大多数 ERC20 的实现不可能失败，除非授权不足或转账的金额太多，大多数开发者已经习惯于忽略 ERC20 代币的返回值，并假设一个失败的 trasfer 将被回退。

坦率地说，如果你只与一个你知道其行为的受信任的 ERC20 代币打交道，这并不重要。但在处理任意的 ERC20 代币时，必须考虑到这种行为上的差异。

在许多合约中都有一个隐含的期望，即失败的转账应该总是回退，而不是返回错误，因为大多数 ERC20 代币没有返回错误的机制，所以这导致了很多混乱。

使这个问题更加复杂的是，一些 ERC20 代币并不遵循返回 true 的协议，特别是 Tether。一些代币在转账失败后会回退，这将导致回退的结果冒泡到调用者。因此，一些库包裹了 ERC20 代币的转账调用，以回退恢复并返回一个布尔值。下面是一些实现方法：

参考：Openzeppelin SafeTransfer 及 Solady SafeTransfer （大大地提高了 Gas 效率）

ERC20: 地址投毒

这不是一个智能合约的漏洞，但为了完整起见，我们在这里提到它。

转账零代币是 ERC20 规范所允许的。这可能会导致前端应用程序的混乱，并可能欺骗用户，让他们错误的以为他们最近将代币发送给了某地址。Metamask 在这个线程中有更多关于这个问题的内容。

ERC20: 查看代码，规避跑路

(在 web3 术语中，"rugged"意味着''跑路", 直译是"从你脚下拉出地毯" 。)

没有什么能阻止有人在 ERC20 代币上添加函数，让他们随意创建、转账和销毁代币--或自毁或升级。所以从根本上说，ERC20 代币的 "无需信任" 程度是有限制的。

借贷协议中的逻辑错误

当考虑到基于 DeFi 协议的借贷如何被破坏时，考虑在软件层面传播的 bug 并影响商业逻辑层面是很有帮助的。形成和完成一个债券合约有很多步骤。这里有一些需要考虑的攻击向量。

贷款人损失的方式

- 使到期本金减少（可能为零）而不进行任何支付的漏洞。
- 当贷款没有偿还或抵押物降到阈值以下时，买方的抵押物不能被清算。
- 如果协议有一个转移债务所有权的机制，这可能是一个从贷款人那里偷取债券的方式。
- 贷款本金或付款的到期日被不适当地移到以后的日期。

借款人损失的方式

- 偿还本金时没有减少本金债务的 bug。
- 一个 bug 或 gas 攻击使用户无法进行支付。
- 本金或利率被非法提高。
- 预言机的操纵导致抵押物贬值。
- 贷款本金或付款的到期日被不适当地移到一个较早的日期。

如果抵押品从协议中被抽走，那么贷款人和借款人都会损失，因为借款人没有动力去偿还贷款，而借款人则会损失本金。

正如上面所看到的，DeFi 协议被 "黑 "的范围比从协议中抽走一堆钱（通常成为新闻的那类事件）要多得多。

抵押（staking）协议中的漏洞

成为新闻的那种黑客是抵押协议被黑掉数百万美元，但这并不是唯一要面对的问题，抵押协议可能面临的问题有：

- 奖励能否延迟支付，或过早地被索取？
- 奖励能否被不适当地减少或增加？在更糟糕的情况下，能否阻止用户获得任何奖励？
- 人们能否索取不属于他们的本金或奖励，在最坏的情况下，会耗尽协议所有资金？
- 存放的资产会不会被卡在协议中（部分或全部），或被不适当地延迟提取？
- 相反，如果质押需要时间承诺，用户是否可以在承诺时间之前提取？
- 如果支付的是不同的资产或货币，其价值是否可以在相关的智能合约范围内被操纵？如果协议 mint 自己的代币来奖励流动性提供者或质押者，这一点是相关的。
- 如果存在预期和披露出的本金损失的风险因素，这种风险是否可以被不适当地操纵？
- 协议的关键参数是否有管理、中心化或治理风险？
- 需要关注的关键是代码中涉及 "资金退出 "部分的代码。

还有一个 "资金入口 "的漏洞也要寻找。

- 有权参与协议中的资产抵押的用户能否被不适当地阻止？

用户收到的奖励有一个隐含的风险回报和一个预期的资金时间价值。明确这些假设是什么，以及协议会怎样偏离预期是很有帮助的。

未检查的返回值

有两种方法来调用外部智能合约：1）用接口定义调用函数；2）使用.call 方法。如下图所示：

```
1 contract A {
```

```
    uint256 public x;

    function setx(uint256 _x) external {
        require(_x > 10, "x must be bigger than 10");
        x = _x;
    }
}

interface IA {
    function setx(uint256 _x) external;
}

contract B {
    function setXV1(IA a, uint256 _x) external {
        a.setx(_x);
    }

    function setXV2(address a, uint256 _x) external {
        (bool success, ) =
            a.call(abi.encodeWithSignature("setx(uint256)", _x));
        // success is not checked!
    }
}
```

在合约 B 中，如果 _x 小于 10，setXV2 会默默地失败。当一个函数通过.call 方法被调用时，被调用者可以回退，但父函数不会回退。必须检查返回成功的值，并且代码行为必须相应地分支。

msg.value 在一个循环中

在循环中使用 msg.value 是很危险的，因为这可能会让发起者 重复使用 msg.value。

这种情况可能会出现在 payable 的 multicalls 中。Multicalls 使用户能够提交一个交易列表，以避免重复支付 21,000 的 Gas 交易费。然而，msg.value 在通过函数循环执行时被 "重复使用"，有可能使用户双花。

这就是 Opyn Hack 的根本原因。

私有变量

私有变量在区块链上仍然是可见的，所以敏感信息不应该被存储在那里。如果它们不能被访问，验证者如何能够处理取决于其值的交易？私有变量不能从外部的 Solidity 合约中读取，但它们可以使用以太坊客户端在链外读取。

要读取一个变量，你需要知道它的存储槽。在下面的例子中，myPrivateVar 的存储槽是 0。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract PrivateVarExample {
    uint256 private myPrivateVar;

    constructor(uint256 _initialValue) {
        myPrivateVar = _initialValue;
    }
}
```

下面是读取已部署的智能合约的私有变量的 javascript 代码

```
const Web3 = require("web3");
const PRIVATE_VAR_EXAMPLE_ADDRESS = "0x123...";

async function readPrivateVar() {
  const web3 = new Web3("http://localhost:8545");
  const storageSlot = 0;
  const privateVarValue = await web3.eth.getStorageAt(
    PRIVATE_VAR_EXAMPLE_ADDRESS,
    storageSlot
  );

  console.log("Value of private variable 'myPrivateVar':",
  web3.utils.hexToNumberString(privateVarValue));
}

readPrivateVar();
```

不安全的代理调用

委托调用（Delegatecall）不应该被用于不受信任的合约，因为它把所有的控制权都交给了委托接受者。在这个例子中，不受信任的合约偷走了合约中所有的以太币。

```
contract UntrustedDelegateCall {
    constructor() payable {
        require(msg.value == 1 ether);
    }

    function doDelegateCall(address _delegate, bytes calldata data) public {
        (bool ok, ) = _delegate.delegatecall(data);
        require(ok, "delegatecall failed");
    }

    }

    contract StealEther {
        function steal() public {
            // you could also selfdestruct here
            // if you really wanted to be mean
            (bool ok,) =
                tx.origin.call{value: address(this).balance}("");
            require(ok);
        }

        function attack(address victim) public {
            UntrustedDelegateCall(victim).doDelegateCall(
                address(this),
                abi.encodeWithSignature("steal()"));
        }
}
```

升级与代理有关的 bug

我们无法在一个章节中对这个话题进行公正的解释。大多数升级错误通常可以通过使用 Openzeppelin 的 hardhat 插件和阅读它所保护的问题来避免出错。

作为一个快速的总结，以下是与智能合约升级有关的问题：

- 自毁（self-destruct）和委托调用（delegatecall）不应该在执行合约中使用。
- 必须注意在升级过程中，存储变量不能相互覆盖
- 在执行合约中应避免调用外部库，因为不可能预测它们会如何影响存储访问。
- 部署者决不能忽视调用初始化函数
- 在基类合约中没有包括间隙（gap）变量，以防止在基类合约中加入新的变量时发生存储碰撞（这由 hardhat 插件自动处理）。
- 不可变（immutable）变量中的值在升级时不会被保留
- 非常不鼓励在构造函数中做任何事情，因为未来的升级必须执行相同的构造函数逻辑以保持兼容性。

智能合约安全 (四) 签名伪造与篡改

数字签名在智能合约的背景下有两种用途：

- 使得地址能够授权区块链上的一些交易，而不进行实际交易
- 根据预定的地址，向智能合约证明发起者有某种权力去做某事

下面是一个安全使用数字签名的例子，让用户拥有铸造 NFT 的特权：

```
1 import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
2 import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
3
4 contract NFT is ERC721("name", "symbol") {
5     function mint(bytes calldata signature) external {
6         address recovered =
  keccak256(abi.encode(msg.sender)).toEthSignedMessageHash().recover(signature);
7         require(recovered == authorizer, "signature does not match");
8     }
9 }
```

一个典型的例子是 ERC20 中的 Approve 函数。为了授权一个地址从我们的账户中提取一定数量的代币，我们必须进行实际的以太坊交易，这需要花费 Gas。

有时，将数字签名传递给链外的接收者，然后接收者向智能合约提供签名，以证明他们被授权进行交易，这样做更有效率。

ERC20Permit 实现了用数字签名进行审批。该函数描述如下:

```
function permit(address owner,
    address spender,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public
```

而不是发送一个实际的授权交易，所有者可以为花费者 "签署"授权（连同一个截止日期）。然后，被授权的花费者可以用提供的签署信息作为参数调用 permint 函数。

签名的剖析

你会经常看到 v、r 和 s 这些变量。它们在 solidity 中分别用 uint8、bytes32 和 bytes32 的数据类型表示。有时，签名被表示为一个 65 字节的数组，它是所有这些值连接起来的，如 abi.encodePacked(r, s, v)；

签名的另外两个基本组成部分是消息哈希值（32 字节）和签名地址。这个序列看起来像这样

1. 私钥（privKey）用于生成一个公共地址（ethAddress）。
2. 一个智能合约预先存储地址 ethAddress

3. 一个链外用户对一个消息进行 Hash，并对 Hash 值进行签名。这就产生了一对 msgHash 和签名（r, s, v）。
4. 智能合约收到一条消息，对其进行散列以产生 msgHash，然后将其与(r, s, v)相结合，看得出什么地址。
5. 如果地址与 ethAddress 匹配，则签名有效（在某些假设下，我们很快就会看到！）。
6. 智能合约使用步骤 4 中的预编译合约 ecrecover 来完成我们所说的组合，并获得地址。

在这个过程中，有很多步骤都会出现问题。

签名：ecrecover 返回地址(0)，当地址无效时，不会被回退

如果一个未初始化的变量与 ecrecover 的输出相比较，这可能导致漏洞。

这段代码有漏洞：

```
contract InsecureContract {

    address signer;
    // defaults to address(0)
    // who lets us give the beneficiary the airdrop without them// spending gas
    function airdrop(address who, uint256 amount, uint8 v, bytes32 r, bytes32 s) external {

        // 如果签名无效，ecrecover 会返回 address(0)
        require(signer == ecrecover(keccak256(abi.encode(who, amount)), v, r, s), "invalid signature");

        mint(msg.sender, AIRDROP_AMOUNT);
    }
}
```

签名重放

签名重放发生在合约没有跟踪签名是否先前被使用。在下面的代码中，我们修复了之前的问题，但它仍然不安全。

```
contract InsecureContract {

    address signer;

    function airdrop(address who, uint256 amount, uint8 v, bytes32 r, bytes32 s) external {

        address recovered == ecrecover(keccak256(abi.encode(who, amount)), v, r, s);
        require(recovered != address(0), "invalid signature");
        require(recovered == signer, "recovered signature not equal signer");


        mint(msg.sender, amount);
    }
}
```

人们可以随心所欲地索取空投!

我们可以添加以下几行:

```
bytes memory signature = abi.encodePacked(v, r, s);
require(!used[signature], "signature already used");
## mapping(bytes => bool);
used[signature] = true;
```

唉，这段代码还是不安全啊!

签名的可塑性

给定一个有效的签名，攻击者可以做一些快速的算术来推导出一个不同的签名。然后，攻击者可以"重放"这个修改过的签名。但首先，让我们提供一些代码，证明我们可以从一个有效的签名开始，修改它，并显示新的签名仍然通过。

```
contract Malleable {

    // v = 28
    // r = 0xf8479d94c011613baeffe9239e4ff65e2adbac744c34217ca7d51378e72c5204
    // s = 0x57af17590a914b759c45aaeabaf513d5ef72d7da1bdd19d9f2e1bc371ece5b86
    // m = 0x0000000000000000000000000000000000000000000000000000000000000003
    function foo(bytes calldata msg, uint8 v, bytes32 r, bytes32 s) public pure returns (address, address){
        bytes32 h = keccak256(msg);
        address a = ecrecover(h, v, r, s);


        // 下面是一个数据魔法用来反转签名并创建有效的签名
        // flip s
        bytes32 s2 = bytes32(uint256(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141) - uint256(s));

        // invert v
        uint8 v2;
        require(v == 27 || v == 28, "invalid v");
        v2 = v == 27 ? 28 : 27;

        address b = ecrecover(h, v2, r, s2);

        assert(a == b);
        ## different signatures, same address!;
        return (a, b);
    }
}
```

因此，我们的运行实例仍然是脆弱的。一旦有人提出一个有效的签名，就可以产生镜像签名并绕过使用的签名检查。

```
contract InsecureContract {

    address signer;

    function airdrop(address who, uint256 amount, uint8 v, bytes32 r, bytes32 s) external {

        address recovered == ecrecover(keccak256(abi.encode(who, amount)), v, r, s);
        require(recovered != address(0), "invalid signature");
        require(recovered == signer, "recovered signature not equal signer");

        bytes memory signature = abi.encodePacked(v, r, s);
        require(!used[signature], "signature already used"); // this can be bypassed
        used[signature] = true;

        mint(msg.sender, amount);
    }
}
```

安全签名

在这一点上，你可能想得到一些安全的签名代码，对吗？我们向你推荐我们的在 solidity 中创建签名和在 foundry 中测试签名的教程。但这里是检查清单：

- 使用 openzeppelin 的库来防止可塑性攻击，并还原到零地址的问题
- 不要使用签名作为密码。信息需要包含攻击者不能轻易重复使用的信息（如 msg.sender）。
- 在链上对你所签署的内容进行 Hash
- 使用 nonce 来防止重放攻击。更好的是，遵循 EIP712，这样用户可以看到他们正在签署的内容，并且可以防止签名在合约和不同链之间被重复使用。
- 签名在没有适当的保障措施的情况下可以被伪造或篡改
- 如果没有在链上进行 Hash，上面的攻击可以进一步泛化。在上面的例子中，Hash 是在智能合约中完成的，所以上面的例子不容易被下面的攻击所攻击。

让我们来看看还原签名的代码：

```
// 这个代码是不安全的
function recoverSigner(bytes32 hash, uint8 v, bytes32 r, bytes32 s) public
returns (address signer) {
    require(signer == ecrecover(hash, v, r, s), "signer does not match");
    // more actions
}
```

用户同时提供哈希值和签名。如果攻击者已经从签名者那里看到了一个有效的签名，他们可以简单地重复使用另一个消息的哈希和签名。

这就是为什么在智能合约中对消息进行哈希非常重要，而不是在链外。

要看这个漏洞的操作，请看我们在 Twitter 上发布的 CTF。

签名作为标识符

签名不应该被用来识别用户。由于可塑性，它们不能被认为是唯一的。Msg.sender 有更强的唯一性保证。