以太坊智能合约 —— 最佳安全开发指南

本文翻译自：https://github.com/ConsenSys/smart-contract-best-practices。 为了使语句表达更加贴切，个别地方未按照原文逐字逐句翻译，如有出入请以原文为准。

主要章节如下:

这篇文档旨在为Solidity开发人员提供一些智能合约的安全准则(**security baseline**)。当然也包括智能合约的**安全开发理念、bug赏金计划指南、文档例程以及工具。**

我们邀请社区对该文档提出修改或增补建议，欢迎各种合并请求(Pull Request)。若有相关的文章或者博客的发表，也清将其加入到参考文献中，具体详情请参见我们的社区贡献指南。

更多期待内容

我们欢迎并期待社区开发者贡献以下几个方面的内容：

- Solidity代码测试（包括代码结构，程序框架 以及 常见软件工程测试）
- 智能合约开发经验总结，以及更广泛的基于区块链的开发技巧分享

基本理念

以太坊和其他复杂的区块链项目都处于早期阶段并且有很强的实验性质。因此，随着新的bug和安全漏洞被发现，新的功能不断被开发出来，其面临的安全威胁也是不断变化的。这篇文章对于开发人员编写安全的智能合约来说只是个开始。

开发智能合约需要一个全新的工程思维，它不同于我们以往项目的开发。因为它犯错的代价是巨大的，并且很难像传统软件那样轻易的打上补丁。就像直接给硬件编程或金融服务类软件开发，相比于web开发和移动开发都有更大的挑战。因此，仅仅防范已知的漏洞是不够的，你还需要学习新的开发理念：

- **对可能的错误有所准备**。任何有意义的智能合约或多或少都存在错误。因此你的代码必须能够正确的处理出现的bug和漏洞。始终保证以下规则： - 当智能合约出现错误时，停止合约，（“断路开关”） - 管理账户的资金风险（限制（转账）速率、最大（转账）额度）
  - 有效的途径来进行bug修复和功能提升
- **谨慎发布智能合约**。 尽量在正式发布智能合约之前发现并修复可能的bug。 - 对智能合约进行彻底的测试，并在任何新的攻击手法被发现后及时的测试(包括已经发布的合约) - 从alpha版本在测试网（testnet）上发布开始便提供bug赏金计划

    - 阶段性发布，每个阶段都提供足够的测试

- **保持智能合约的简洁**。复杂会增加出错的风险。
    - 确保智能合约逻辑简洁
    - 确保合约和函数模块化
    - 使用已经被广泛使用的合约或工具（比如，不要自己写一个随机数生成器）
    - 条件允许的话，清晰明了比性能更重要
    - 只在你系统的去中心化部分使用区块链

- **保持更新**。通过下一章节所列出的资源来确保获取到最新的安全进展。
    - 在任何新的漏洞被发现时检查你的智能合约
    - 尽可能快的将使用到的库或者工具更新到最新
    - 使用最新的安全技术

- **清楚区块链的特性**。尽管你先前所拥有的编程经验同样适用于以太坊开发，但这里仍然有些陷阱你需要留意：
    - 特别小心针对外部合约的调用，因为你可能执行的是一段恶意代码然后更改控制流程
    - 清楚你的public function是公开的，意味着可以被恶意调用。（在以太坊上）你的private data也是对他人可见的
    - 清楚gas的花费和区块的gas limit

基本权衡：简单性与复杂性

在评估一个智能合约的架构和安全性时有很多需要权衡的地方。对任何智能合约的建议是在各个权衡点中找到一个平衡点。

从传统软件工程的角度出发：一个理想的智能合约首先需要模块化，能够重用代码而不是重复编写，并且支持组件升级。从智能合约安全架构的角度出发同样如此，模块化和重用被严格审查检验过的合约是最佳策略，特别是在复杂智能合约系统里。

然而，这里有几个重要的例外，它们从合约安全和传统软件工程两个角度考虑，所得到的重要性排序可能不同。当中每一条，都需要针对智能合约系统的特点找到最优的组合方式来达到平衡。

- 固化 vs 可升级
- 庞大 vs 模块化
- 重复 vs 可重用

固化 vs 可升级

在很多文档或者开发指南中，包括该指南，都会强调延展性比如：可终止，可升级或可更改的特性，不过对于智能合约来说，延展性和安全之间是个*基本权衡*。

延展性会增加程序复杂性和潜在的攻击面。对于那些只在特定的时间段内提供有限的功能的智能合约，简单性比复杂性显得更加高效，比如无管治功能，有限短期内使用的代币发行的智能合约系统(governance-fee,finite-time-frame token-sale contracts)。

庞大 vs 模块化

一个庞大的独立的智能合约把所有的变量和模块都放到一个合约中。尽管只有少数几个大家熟知的智能合约系统真的做到了大体量，但在将数据和流程都放到一个合约中还是享有部分优点--比如，提高代码审核(code review)效率。

和在这里讨论的其他权衡点一样，传统软件开发策略和从合约安全角度出发考虑，两者不同主要在对于简单、短生命周期的智能合约；对于更复杂、长生命周期的智能合约，两者策略理念基本相同。

重复 vs 可重用

从软件工程角度看，智能合约系统希望在合理的情况下最大程度地实现重用。 在Solidity中重用合约代码有很多方法。 使用**你拥有的**以前部署的经过验证的智能合约是实现代码重用的最安全的方式。

在以前所拥有已部署智能合约不可重用时重复还是很需要的。 现在Live Libs 和Zeppelin Solidity 正寻求提供安全的智能合约组件使其能够被重用而不需要每次都重新编写。任何合约安全性分析都必须标明重用代码，特别是以前没有建立与目标智能合同系统中处于风险中的资金相称的信任级别的代码。

安全通知

以下这些地方通常会通报在Ethereum或Solidity中新发现的漏洞。安全通告的官方来源是Ethereum Blog，但是一般漏洞都会在其他地方先被披露和讨论。

- Ethereum Blog: The official Ethereum blog
  - Ethereum Blog - Security only: 所有相关博客都带有**Security**标签
- Ethereum Gitter 聊天室
  - Solidity
  - Go-Ethereum
  - CPP-Ethereum
  - Research
- Reddit
- Network Stats

强烈建议你经常浏览这些网站，尤其是他们提到的可能会影响你的智能合约的漏洞。

另外, 这里列出了以太坊参与安全模块相关的核心开发成员, 浏览 bibliography 获取更多信息。

- **Vitalik Buterin**: Twitter, Github, Reddit, Ethereum Blog
- **Dr. Christian Reitwiessner**: Twitter, Github, Ethereum Blog
- **Dr. Gavin Wood**: Twitter, Blog, Github
- **Vlad Zamfir**: Twitter, Github, Ethereum Blog

除了关注核心开发成员，参与到各个区块链安全社区也很重要，因为安全漏洞的披露或研究将通过各方进行。

关于使用Solidity开发的智能合约安全建议

外部调用

尽量避免外部调用

调用不受信任的外部合约可能会引发一系列意外的风险和错误。外部调用可能在其合约和它所依赖的其他合约内执行恶意代码。因此，每一个外部调用都会有潜在的安全威胁，尽可能的从你的智能合约内移除外部调用。当无法完全去除外部调用时，可以使用这一章节其他部分提供的建议来尽量减少风险。

仔细权衡“send()”、“transfer()”、以及“call.value()”

当转账Ether时，需要仔细权衡“someAddress.send()”、“someAddress.transfer()”、和“someAddress.call.value()()”之间的差别。

- `x.transfer(y)` 和 `if (!x.send(y)) throw;` 是等价的。send是transfer的底层实现，建议尽可能直接使用transfer。
- `someAddress.send()` 和 `someAddress.transfer()` 能保证可重入 **安全** 。 尽管这些外部智能合约的函数可以被触发执行，但补贴给外部智能合约的2,300 gas，意味着仅仅只够记录一个event到日志中。
- `someAddress.call.value()()` 将会发送指定数量的Ether并且触发对应代码的执行。被调用的外部智能合约代码将享有所有剩余的gas，通过这种方式转账是很容易有可重入漏洞的，非常 **不安全**。

使用 `send()` 或 `transfer()` 可以通过制定gas值来预防可重入， 但是这样做可能会导致在和合约调用fallback函数时出现问题，由于gas可能不足，而合约的fallback函数执行至少需要2,300 gas消耗。

一种被称为*push* 和*pull*的 机制试图来平衡两者， 在 *push* 部分使用 `send()` 或 `transfer()` ，在*pull* 部分使用 `call.value()()` 。（*译者注：在需要对外未知地址转账Ether时使用 `send()` 或 `transfer()` ，已知明确内部无恶意代码的地址转账Ether使用 `call.value()()` ）

需要注意的是使用 `send()` 或 `transfer()` 进行转账并不能保证该智能合约本身重入安全，它仅仅只保证了这次转账操作时重入安全的。

处理外部调用错误

Solidity提供了一系列在raw address上执行操作的底层方法，比如： `address.call()` ， `address.callcode()` ， `address.delegatecall()` 和 `address.send` 。这些底层方法不会抛出异常(throw)，只是会在遇到错误时返回false。另一方面， *contract calls* （比如， `ExternalContract.doSomething()` ）会自动传递异常，（比如， `doSomething()` 抛出异常，那么 `ExternalContract.doSomething()` 同样会进行 `throw` ）。

如果你选择使用底层方法，一定要检查返回值来对可能的错误进行处理。

```
// bad
someAddress.send(55);
someAddress.call.value(55)(); // this is doubly dangerous, as it will forward all remaining gas and doesn't
someAddress.call.value(100)(bytes4(sha3("deposit()"))); // if deposit throws an exception, the raw call() v

// good
if(!someAddress.send(55)) {
    // Some failure code
}

ExternalContract(someAddress).deposit.value(100);
```

不要假设你知道外部调用的控制流程

无论是使用**raw calls** 或是**contract calls**，如果这个 `ExternalContract` 是不受信任的都应该假设存在恶意代码。即使 `ExternalContract` 不包含恶意代码，但它所调用的其他合约代码可能会包含恶意代码。一个具体的危险例子便是恶意代码可能会劫持控制流程导致竞态。（浏览Race Conditions获取更多关于这个问题的讨论）

对于外部合约优先使用*pull* 而不是*push*

外部调用可能会有意或无意的失败。为了最小化这些外部调用失败带来的损失，通常好的做法是将外部调用函数与其余代码隔离，最终是由收款发起方负责发起调用该函数。这种做法对付款操作尤为重要，比如让用户自己撤回资产而不是直接发送给他们。（*译者注：事先设置需要付给某一方的资产的值，表明接收方可以从当前账户撤回资金的额度，然后由接收方调用当前合约提现函数完成转账*）。（这种方法同时也避免了造成 gas limit相关问题。）

```
// bad
contract auction {
    address highestBidder;
    uint highestBid;

    function bid() payable {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            if (!highestBidder.send(highestBid)) { // if this call consistently fails, no one else can bid
                throw;
            }
        }

       highestBidder = msg.sender;
       highestBid = msg.value;
    }
}

// good
contract auction {
    address highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() payable external {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            refunds[highestBidder] += highestBid; // record the refund that this user can claim
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        if (!msg.sender.send(refund)) {
            refunds[msg.sender] = refund; // reverting state because send failed
        }
    }
}
```

标记不受信任的合约

当你自己的函数调用外部合约时，你的变量、方法、合约接口命名应该表明和他们可能是不安全的。

```
// bad
Bank.withdraw(100); // Unclear whether trusted or untrusted

function makeWithdrawal(uint amount) { // Isn't clear that this function is potentially unsafe
    Bank.withdraw(amount);
}

// good
UntrustedBank.withdraw(100); // untrusted external call
TrustedBank.withdraw(100); // external but trusted bank contract maintained by XYZ Corp

function makeUntrustedWithdrawal(uint amount) {
    UntrustedBank.withdraw(amount);
}
```

使用 `assert()` 强制不变性

当断言条件不满足时将触发断言保护 -- 比如不变的属性发生了变化。举个例子，代币在以太坊上的发行比例，在代币的发行合约里可以通过这种方式得到解决。断言保护经常需要和其他技术组合使用，比如当断言被触发时先挂起合约然后升级。（否则将一直触发断言，你将陷入僵局）

例如：

```
contract Token {
    mapping(address => uint) public balanceOf;
    uint public totalSupply;

    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
        totalSupply += msg.value;
        assert(this.balance >= totalSupply);
    }
}
```

注意断言保护 **不是** 严格意义的余额检测， 因为智能合约可以不通过 `deposit()` 函数被 强制发送Ether!

正确使用 `assert()` 和 `require()`

在Solidity 0.4.10 中 `assert()` 和 `require()` 被加入。 `require(condition)` 被用来验证用户的输入，如果条件不满足便会抛出异常，应当使用它验证所有用户的输入。 `assert(condition)` 在条件不满足也会抛出异常，但是最好只用于固定变量：内部错误或你的智能合约陷入无效的状态。遵循这些范例，使用分析工具来验证永远不会执行这些无效操作码：意味着代码中不存在任何不变量，并且代码已经正式验证。

小心整数除法的四舍五入

所有整数除数都会四舍五入到最接近的整数。 如果您需要更高精度，请考虑使用乘数，或存储分子和分母。

（将来Solidity会有一个fixed-point类型来让这一切变得容易。）

```
// bad
uint x = 5 / 2; // Result is 2, all integer divison rounds DOWN to the nearest integer

// good
uint multiplier = 10;
uint x = (5 * multiplier) / 2;

uint numerator = 5;
uint denominator = 2;
```

记住Ether可以被强制发送到账户

谨慎编写用来检查账户余额的不变量。

攻击者可以强制发送wei到任何账户，而且这是不能被阻止的（即使让fallback函数 `throw` 也不行）

攻击者可以仅仅使用1 wei来创建一个合约，然后调用 `selfdestruct(victimAddress)` 。在 `victimAddress` 中没有代码被执行，所以这是不能被阻止的。

不要假设合约创建时余额为零

攻击者可以在合约创建之前向合约的地址发送wei。合约不能假设它的初始状态包含的余额为零。浏览issue 61 获取更多信息。

记住链上的数据是公开的

许多应用需要提交的数据是私有的，直到某个时间点才能工作。游戏（比如，链上游戏rock-paper-scissors（石头剪刀布））和拍卖机（比如，sealed-bid second-price auctions）是两个典型的例子。如果你的应用存在隐私保护问题，一定要避免过早发布用户信息。

例如：

- 在游戏石头剪刀布中，需要参与游戏的双方提交他们"行动计划"的hash值，然后需要双方随后提交他们的行动计划；如果双方的"行动计划"和先前提交的hash值对不上则抛出异常。
- 在拍卖中，要求玩家在初始阶段提交其所出价格的hash值（以及超过其出价的保证金），然后在第二阶段提交他们所出价格的资金。
- 当开发一个依赖随机数生成器的应用时，正确的顺序应当是（1）玩家提交行动计划，（2）生成随机数，（3）玩家支付。产生随机数是一个值得研究的领域；当前最优的解决方案包括比特币区块头（通过http://btcrelay.org验证），hash-commit-reveal方案（比如，一方产生number后，将其散列值提交作为对这个number的"提交"，然后在随后再暴露这个number本身）和 RANDAO。
- 如果你正在实现频繁的批量拍卖，那么hash-commit机制也是个不错的选择。

权衡Abstract合约和Interfaces

Interfaces和Abstract合约都是用来使智能合约能更好的被定制和重用。Interfaces是在Solidity 0.4.11中被引入的，和Abstract合约很像但是不能定义方法只能申明。Interfaces存在一些限制比如不能够访问storage或者从其他Interfaces那继承，通常这些使Abstract合约更实用。尽管如此，Interfaces在实现智能合约之前的设计智能合约阶段仍然有很大用处。另外，需要注意的是如果一个智能合约从另一个Abstract合约继承而来那么它必须实现所有Abstract合约内的申明并未实现的函数，否则它也会成为一个Abstract合约。

在双方或多方参与的智能合约中，参与者可能会"脱机离线"后不再返回

不要让退款和索赔流程依赖于参与方执行的某个特定动作而没有其他途径来获取资金。比如，在石头剪刀布游戏中，一个常见的错误是在两个玩家提交他们的行动计划之前不要付钱。然而一个恶意玩家可以通过一直不提交它的行动计划来使对方蒙受损失 -- 事实上，如果玩家看到其他玩家泄露的行动计划然后决定他是否会损失（译者注：发现自己输了），那么他完全有理由不再提交他自己的行动计划。这些问题也同样会出现在通道结算。当这些情形出现导致问题后：（1）提供一种规避非参与者和参与者的方式，可能通过设置时间限制，和（2）考虑为参与者提供额外的经济激励，以便在他们应该这样做的所有情况下仍然提交信息。

使Fallback函数尽量简单

Fallback函数在合约执行消息发送没有携带参数（或当没有匹配的函数可供调用）时将会被调用，而且当调用 `.send()` or `.transfer()` 时，只会有2,300 gas 用于失败后fallback函数的执行（*译者注：合约收到Ether也会触发fallback函数执行*）。如果你希望能够监听 `.send()` 或 `.transfer()` 接收到Ether，则可以在fallback函数中使用event（译者注：让客户端监听相应事件做相应处理）。谨慎编写fallback函数以免gas不够用。

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

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

function() payable { LogDepositReceived(msg.sender); }
```

明确标明函数和状态变量的可见性

明确标明函数和状态变量的可见性。函数可以声明为 `external` ， `public` ， `internal` 或 `private` 。分清楚它们之间的差异， 例如 `external` 可能已够用而不是使用 `public` 。对于状态变量， `external` 是不可能的。明确标注可见性将使得更容易避免关于谁可以调用该函数或访问变量的错误假设。

```
// bad
uint x; // the default is private for state variables, but it should be made explicit
function buy() { // the default is public
    // public code
}

// good
uint private y;
function buy() external {
    // only callable externally
}

function utility() public {
    // callable externally, as well as internally: changing this code requires thinking about both cases.
}

function internalAction() internal {
    // internal code
}
```

将程序锁定到特定的编译器版本

智能合约应该应该使用和它们测试时使用最多的编译器相同的版本来部署。锁定编译器版本有助于确保合约不会被用于最新的可能还有bug未被发现的编译器去部署。智能合约也可能会由他人部署，而pragma标明了合约作者希望使用哪个版本的编译器来部署合约。

```
// bad
pragma solidity ^0.4.4;


// good
pragma solidity 0.4.4;
```

（*译者注：这当然也会付出兼容性的代价*）

小心分母为零 (Solidity < 0.4)

早于0.4版本, 当一个数尝试除以零时，Solidity 返回zero 并没有 `throw` 一个异常。确保你使用的Solidity版本至少为 0.4。

区分函数和事件

为了防止函数和事件（Event）产生混淆，命名一个事件使用大写并加入前缀（我们建议**LOG**）。对于函数， 始终以小写字母开头，构造函数除外。

```
// bad
event Transfer() {}
function transfer() {}

// good
event LogTransfer() {}
function transfer() external {}
```

使用Solidity更新的构造器

更合适的构造器/别名，如 `selfdestruct` （旧版本为'suicide ）和 keccak256 （旧版本为 sha3 ）。 像 require(msg.sender.send(1 ether))``的模式也可以简化为使用 `transfer()` ，如`msg.sender.transfer(1 ether)`。

已知的攻击

竞态*

调用外部契约的主要危险之一是它们可以接管控制流，并对调用函数意料之外的数据进行更改。 这类bug有多种形式，导致DAO崩溃的两个主要错误都是这种错误。

重入

这个版本的bug被注意到是其可以在第一次调用这个函数完成之前被多次重复调用。对这个函数不断的调用可能会造成极大的破坏。

```
// INSECURE
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    if (!(msg.sender.call.value(amountToWithdraw)())) { throw; } // At this point, the caller's code is exe
    userBalances[msg.sender] = 0;
}
```

（译者注：使用msg.sender.call.value()()）传递给fallback函数可用的气是当前剩余的所有气，在这里，假如从你账户执行提现操作的恶意合约的fallback函数内递归调用你的withdrawBalance()便可以从你的账户转走更多的币。）

可以看到当调msg.sender.call.value()()时，并没有将userBalances[msg.sender] 清零，于是在这之前可以成功递归调用很多次withdrawBalance()函数。 一个非常相像的bug便是出现在针对 DAO 的攻击。

在给出来的例子中，最好的方法是 使用 `send()` 而不是 `call.value()()` 。这将避免多余的代码被执行。

然而，如果你没法完全移除外部调用，另一个简单的方法来阻止这个攻击是确保你在完成你所有内部工作之前不要进行外部调用：

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    if (!(msg.sender.call.value(amountToWithdraw)())) { throw; } // The user's balance is already 0, so fut
}
```

注意如果你有另一个函数也调用了 `withdrawBalance()` ， 那么这里潜在的存在上面的攻击，所以你必须认识到任何调用了不受信任的合约代码的合约也是不受信任的。继续浏览下面的相关潜在威胁解决办法的讨论。

跨函数竞态

攻击者也可以使用两个共享状态变量的不同的函数来进行类似攻击。

```
// INSECURE
mapping (address => uint) private userBalances;

function transfer(address to, uint amount) {
    if (userBalances[msg.sender] >= amount) {
       userBalances[to] += amount;
       userBalances[msg.sender] -= amount;
    }
}

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    if (!(msg.sender.call.value(amountToWithdraw)())) { throw; } // At this point, the caller's code is exe
    userBalances[msg.sender] = 0;
}
```

着这个例子中，攻击者在他们外部调用 `withdrawBalance` 函数时调用 `transfer()` ， 如果这个时候 `withdrawBalance` 还没有执行到 `userBalances[msg.sender] = 0;` 这里，那么他们的余额就没有被清零，那么他们就能够调用 `transfer()` 转走代币尽管他们其实已经收到了代币。这个弱点也可以被用到对DAO的攻击。

同样的解决办法也会管用，在执行转账操作之前先清零。也要注意在这个例子中所有函数都是在同一个合约内。然而，如果这些合约共享了状态，同样的bug也可以发生在跨合约调用中。

竞态解决办法中的陷阱

由于竞态既可以发生在跨函数调用，也可以发生在跨合约调用，任何只是避免重入的解决办法都是不够的。

作为替代，我们建议首先应该完成所有内部的工作然后再执行外部调用。这个规则可以避免竞态发生。然而，你不仅应该避免过早调用外部函数而且应该避免调用那些也调用了外部函数的外部函数。例如，下面的这段代码是不安全的：

```
// INSECURE
mapping (address => uint) private userBalances;
mapping (address => bool) private claimedBonus;
mapping (address => uint) private rewardsForA;

function withdraw(address recipient) public {
    uint amountToWithdraw = userBalances[recipient];
    rewardsForA[recipient] = 0;
    if (!(recipient.call.value(amountToWithdraw)())) { throw; }
}

function getFirstWithdrawalBonus(address recipient) public {
    if (claimedBonus[recipient]) { throw; } // Each recipient should only be able to claim the bonus once

    rewardsForA[recipient] += 100;
    withdraw(recipient); // At this point, the caller will be able to execute getFirstWithdrawalBonus agair
    claimedBonus[recipient] = true;
}
```

尽管 `getFirstWithdrawalBonus()` 没有直接调用外部合约，但是它调用的 `withdraw()` 却会导致竞态的产生。在这里你不应该认为 `withdraw()` 是受信任的。

```
mapping (address => uint) private userBalances;
mapping (address => bool) private claimedBonus;
mapping (address => uint) private rewardsForA;

function untrustedWithdraw(address recipient) public {
    uint amountToWithdraw = userBalances[recipient];
    rewardsForA[recipient] = 0;
    if (!(recipient.call.value(amountToWithdraw)())) { throw; }
}

function untrustedGetFirstWithdrawalBonus(address recipient) public {
    if (claimedBonus[recipient]) { throw; } // Each recipient should only be able to claim the bonus once

    claimedBonus[recipient] = true;
    rewardsForA[recipient] += 100;
    untrustedWithdraw(recipient); // claimedBonus has been set to true, so reentry is impossible
}
```

除了修复bug让重入不可能成功，不受信任的函数也已经被标记出来 。同样的情景：
`untrustedGetFirstWithdrawalBonus()` 调用 `untrustedWithdraw()`，而后者调用了外部合约，因此在这里 `untrustedGetFirstWithdrawalBonus()` 是不安全的。

另一个经常被提及的解决办法是（*译者注：像传统多线程编程中一样*）使用mutex。它会"lock" 当前状态，只有锁的当前拥有者能够更改当前状态。一个简单的例子如下：

```
// Note: This is a rudimentary example, and mutexes are particularly useful where there is substantial logi
mapping (address => uint) private balances;
bool private lockBalances;

function deposit() payable public returns (bool) {
    if (!lockBalances) {
        lockBalances = true;
        balances[msg.sender] += msg.value;
        lockBalances = false;
        return true;
    }
    throw;
}

function withdraw(uint amount) payable public returns (bool) {
    if (!lockBalances && amount > 0 && balances[msg.sender] >= amount) {
        lockBalances = true;

        if (msg.sender.call(amount)()) { // Normally insecure, but the mutex saves it
          balances[msg.sender] -= amount;
```

```
        }

        lockBalances = false;
        return true;
    }

    throw;
}
```

如果用户试图在第一次调用结束前第二次调用 `withdraw()`，将会被锁住。 这看上去很有效果，但当你使用多个合约互相交互时问题变得严峻了。 下面是一段不安全的代码：

```
// INSECURE
contract StateHolder {
    uint private n;
    address private lockHolder;

    function getLock() {
        if (lockHolder != 0) { throw; }
        lockHolder = msg.sender;
    }

    function releaseLock() {
        lockHolder = 0;
    }

    function set(uint newState) {
        if (msg.sender != lockHolder) { throw; }
        n = newState;
    }
}
```

攻击者可以只调用 `getLock()`，然后就不再调用 `releaseLock()`。如果他们真这样做，那么这个合约将会被永久锁住，任何接下来的操作都不会发生了。如果你使用mutexs来避免竞态，那么一定要确保没有地方能够打断锁的进程或绝不释放锁。（这里还有一个潜在的威胁，比如死锁和活锁。在你决定使用锁之前最好大量阅读相关文献（*译者注：这是真的，传统的在多线程环境下对锁的使用一直是个容易犯错的地方*））

* 有些人可能会发反对使用该术语 *竞态*，因为以太坊并没有真正意思上实现并行执行。然而在逻辑上依然存在对资源的竞争，同样的陷阱和潜在的解决方案。

交易顺序依赖(TOD) / 前面的先运行

以上是涉及攻击者在**单个交易**内执行恶意代码产生竞态的示例。接下来演示在区块链本身运作原理导致的竞态：（同一个block内的）交易顺序很容易受到操纵。

由于交易在短暂的时间内会先存放到mempool中，所以在矿工将其打包进block之前，是可以知道会发生什么动作的。这对于一个去中心化的市场来说是麻烦的，因为可以查看到代币的交易信息，并且可以在它被打包进block之前改变交易顺序。避免这一点很困难，因为它归结为具体的合同本身。例如，在市场上，最好实施批量拍卖（这也可以防止高频交易问题）。 另一种使用预提交方案的方法（“我稍后会提供详细信息”）。

时间戳依赖

请注意，块的时间戳可以由矿工操纵，并且应考虑时间戳的所有直接和间接使用。 **区块数量**和**平均出块时间**可用于估计时间，但这不是区块时间在未来可能改变（例如Casper期望的更改）的证明。

```
uint someVariable = now + 1;

if (now % 2 == 0) { // the now can be manipulated by the miner

}

if ((someVariable - 100) % 2 == 0) { // someVariable can be manipulated by the miner

}
```

整数上溢和下溢

这里大概有 20关于上溢和下溢的例子。

考虑如下这个简单的转账操作：

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

// INSECURE
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance */
    if (balanceOf[msg.sender] < _value)
        throw;
    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}

// SECURE
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance and for overflows */
    if (balanceOf[msg.sender] < _value || balanceOf[_to] + _value < balanceOf[_to])
        throw;

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}
```

如果余额到达**uint**的最大值（2^256），便又会变为0。应当检查这里。溢出是否与之相关取决于具体的实施方式。想想uint值是否有机会变得这么大或和谁会改变它的值。如果任何用户都有权利更改uint的值，那么它将更容易受到攻击。如果只有管理员能够改变它的值，那么它可能是安全的，因为没有别的办法可以跨越这个限制。

对于下溢同样的道理。如果一个uint别改变后小于0，那么将会导致它下溢并且被设置成为最大值（2^256）。

对于较小数字的类型比如uint8、uint16、uint24等也要小心：他们更加容易达到最大值。

通过(Unexpected) Throw发动DoS

考虑如下简单的智能合约：

```
// INSECURE
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        if (msg.value <= highestBid) { throw; }

        if (!currentLeader.send(highestBid)) { throw; } // Refund the old leader, and throw if it fails

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

当有更高竞价时，它将试图退款给曾经最高竞价人，如果退款失败则会抛出异常。这意味着，恶意投标人可以成为当前最高竞价人，同时确保对其地址的任何退款**始终**失败。这样就可以阻止任何人调用“bid()”函数，使自己永远保持领先。建议向之前所说的那样建立基于pull的支付系统 。

另一个例子是合约可能通过数组迭代来向用户支付（例如，众筹合约中的支持者）时。 通常要确保每次付款都成功。 如果没有，应该抛出异常。 问题是，如果其中一个支付失败，您将恢复整个支付系统，这意味着该循环将永远不会完成。 因为一个地址没有转账成功导致其他人都没得到报酬。

```
address[] private refundAddresses;
mapping (address => uint) public refunds;

// bad
function refundAll() public {
    for(uint x; x < refundAddresses.length; x++) { // arbitrary length iteration based on how many addresse
        if(refundAddresses[x].send(refunds[refundAddresses[x]])) {
            throw; // doubly bad, now a single failure on send will hold up all funds
```

```
        }
    }
}
```

再一次强调，同样的解决办法： 优先使用pull 而不是push支付系统。

通过区块Gas Limit发动DoS

在先前的例子中你可能已经注意到另一个问题：一次性向所有人转账，很可能会导致达到以太坊区块gas limit的上限。以太坊规定了每一个区块所能花费的gas limit，如果超过你的交易便会失败。

即使没有故意的攻击，这也可能导致问题。然而，最为糟糕的是如果gas的花费被攻击者操控。在先前的例子中，如果攻击者增加一部分收款名单，并设置每一个收款地址都接收少量的退款。这样一来，更多的gas将会被花费从而导致达到区块gas limit的上限，整个转账的操作也会以失败告终。

又一次证明了 优先使用pull 而不是push支付系统。

如果你实在必须通过遍历一个变长数组来进行转账，最好估计完成它们大概需要多少个区块以及多少笔交易。然后你还必须能够追踪得到当前进行到哪以便当操作失败时从那里开始恢复，举个例子：

```
struct Payee {
    address addr;
    uint256 value;
}
Payee payees[];
uint256 nextPayeeIndex;

function payOut() {
    uint256 i = nextPayeeIndex;
    while (i < payees.length && msg.gas > 200000) {
      payees[i].addr.send(payees[i].value);
      i++;
    }
    nextPayeeIndex = i;
}
```

如上所示，你必须确保在下一次执行 `payOut()` 之前另一些正在执行的交易不会发生任何错误。如果必须，请使用上面这种方式来处理。

~~Call Depth攻击~~

由于EIP 150 进行的硬分叉，Call Depth攻击已经无法实施* （由于以太坊限制了Call Depth最大为1024，确保了在达到最大深度之前gas都能被正确使用）

软件工程开发技巧

正如我们先前在基本理念 章节所讨论的那样，避免自己遭受已知的攻击是不够的。由于在链上遭受攻击损失是巨大的，因此你还必须改变你编写软件的方式来抵御各种攻击。

我们倡导“时刻准备失败"，提前知道你的代码是否安全是不可能的。然而，我们可以允许合约以可预知的方式失败，然后最小化失败带来的损失。本章将带你了解如何为可预知的失败做准备。

注意：当你向你的系统添加新的组件时总是伴随着风险的。一个不良设计本身会成为漏洞-一些精心设计的组件在交互过程中同样会出现漏洞。仔细考虑你在合约里使用的每一项技术，以及如何将它们整合共同创建一个稳定可靠的系统。

升级有问题的合约

如果代码中发现了错误或者需要对某些部分做改进都需要更改代码。在以太坊上发现一个错误却没有办法处理他们是太多意义的。

关于如何在以太坊上设计一个合约升级系统是一个正处于积极研究的领域，在这篇文章当中我们没法覆盖所有复杂的领域。然而，这里有两个通用的基本方法。最简单的是专门设计一个注册合约，在注册合约中保存最新版合约的地址。对于合约使用者来说更能实现无缝衔接的方法是设计一个合约，使用它转发调用请求和数据到最新版的合约。

无论采用何种技术，组件之间都要进行模块化和良好的分离，由此代码的更改才不会破坏原有的功能，造成孤儿数据，或者带来巨大的成本。 尤其是将复杂的逻辑与数据存储分开，这样你在使用更改后的功能时不必重新创建所有数据。

当需要多方参与决定升级代码的方式也是至关重要的。根据你的合约，升级代码可能会需要通过单个或多个受信任方参与投票决定。如果这个过程会持续很长时间，你就必须要考虑是否要换成一种更加高效的方式以防止遭受到攻击，例如紧急停止或断路器。

Example 1：使用注册合约存储合约的最新版本

在这个例子中，调用没有被转发，因此用户必须每次在交互之前都先获取最新的合约地址。

```
contract SomeRegister {
    address backendContract;
    address[] previousBackends;
    address owner;

    function SomeRegister() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        if (msg.sender != owner) {
            throw;
        }
        _;
    }

    function changeBackend(address newBackend) public
    onlyOwner()
    returns (bool)
    {
        if(newBackend != backendContract) {
            previousBackends.push(backendContract);
            backendContract = newBackend;
            return true;
        }

        return false;
    }
}
```

这种方法有两个主要的缺点：

1、用户必须始终查找当前合约地址，否则任何未执行此操作的人都可能会使用旧版本的合约 2、在你替换了合约后你需要仔细考虑如何处理原合约中的数据

另外一种方法是设计一个用来转发调用请求和数据到最新版的合约：

例2： 使用 DELEGATECALL 转发数据和调用

```
contract Relay {
    address public currentVersion;
    address public owner;

    modifier onlyOwner() {
        if (msg.sender != owner) {
            throw;
        }
        _;
    }

    function Relay(address initAddr) {
        currentVersion = initAddr;
        owner = msg.sender; // this owner may be another contract with multisig, not a single contract
owner
    }

    function changeContract(address newVersion) public
    onlyOwner()
    {
        currentVersion = newVersion;
    }

    function() {
        if(!currentVersion.delegatecall(msg.data)) throw;
```

```
    }
}
```

这种方法避免了先前的问题，但也有自己的问题。它使得你必须在合约里小心的存储数据。如果新的合约和先前的合约有不同的存储层，你的数据可能会被破坏。另外，这个例子中的模式没法从函数里返回值，只负责转发它们，由此限制了它的适用性。（这里有一个更复杂的实现 想通过内联汇编和返回大小的注册表来解决这个问题）

无论你的方法如何，重要的是要有一些方法来升级你的合约，否则当被发现不可避免的错误时合约将没法使用。

断路器（暂停合约功能）

由于断路器在满足一定条件时将会停止执行，如果发现错误时可以使用断路器。例如，如果发现错误，大多数操作可能会在合约中被挂起，这是唯一的操作就是撤销。你可以授权给任何你受信任的一方，提供给他们触发断路器的能力，或者设计一个在满足某些条件时自动触发某个断路器的程序规则。

例如：

```
bool private stopped = false;
address private owner;

modifier isAdmin() {
    if(msg.sender != owner) {
        throw;
    }
    _;
}

function toggleContractActive() isAdmin public
{
    // You can add an additional modifier that restricts stopping a contract to be based on another
action, such as a vote of users
    stopped = !stopped;
}

modifier stopInEmergency { if (!stopped) _; }
modifier onlyInEmergency { if (stopped) _; }

function deposit() stopInEmergency public
{
    // some code
}

function withdraw() onlyInEmergency public
{
    // some code
}
```

速度碰撞（延迟合约动作）

速度碰撞使动作变慢，所以如果发生了恶意操作便有时间恢复。例如，The DAO 从发起分割DAO请求到真正执行动作需要27天。这样保证了资金在此期间被锁定在合约里，增加了系统的可恢复性。在DAO攻击事件中，虽然在速度碰撞给定的时间段内没有有效的措施可以采取，但结合我们其他的技术，它们是非常有效的。

例如：

```
struct RequestedWithdrawal {
    uint amount;
    uint time;
}

mapping (address => uint) private balances;
mapping (address => RequestedWithdrawal) private requestedWithdrawals;
uint constant withdrawalWaitPeriod = 28 days; // 4 weeks

function requestWithdrawal() public {
    if (balances[msg.sender] > 0) {
        uint amountToWithdraw = balances[msg.sender];
        balances[msg.sender] = 0; // for simplicity, we withdraw everything;
        // presumably, the deposit function prevents new deposits when withdrawals are in progress

        requestedWithdrawals[msg.sender] = RequestedWithdrawal({
```

```
            amount: amountToWithdraw,
            time: now
        });
    }
}

function withdraw() public {
    if(requestedWithdrawals[msg.sender].amount > 0 && now > requestedWithdrawals[msg.sender].time +
withdrawalWaitPeriod) {
        uint amountToWithdraw = requestedWithdrawals[msg.sender].amount;
        requestedWithdrawals[msg.sender].amount = 0;

        if(!msg.sender.send(amountToWithdraw)) {
            throw;
        }
    }
}
```

速率限制

速率限制暂停或需要批准进行实质性更改。 例如，只允许存款人在一段时间内提取总存款的一定数量或百分比（例如，1天内最多100个ether） - 该时间段内的额外提款可能会失败或需要某种特别批准。 或者将速率限制做在合约级别，合约期限内只能发出发送一定数量的代币。

浏览例程

合约发布

在将大量资金放入合约之前，合约应当进行大量的长时间的测试。

至少应该：

- 拥有100%测试覆盖率的完整测试套件（或接近它）
- 在自己的testnet上部署
- 在公共测试网上部署大量测试和错误奖励
- 彻底的测试应该允许各种玩家与合约进行大规模互动
- 在主网上部署beta版以限制风险总额

自动弃用

在合约测试期间，你可以在一段时间后强制执行自动弃用以阻止任何操作继续进行。例如，alpha版本的合约工作几周，然后自动关闭所有除最终退出操作的操作。

```
modifier isActive() {
    if (block.number > SOME_BLOCK_NUMBER) {
        throw;
    }
    _;
}

function deposit() public
isActive() {
    // some code
}

function withdraw() public {
    // some code
}
```

#####限制每个用户/合约的Ether数量

在早期阶段，你可以限制任何用户（或整个合约）的Ether数量 - 以降低风险。

Bug赏金计划

运行赏金计划的一些提示：

- 决定赏金以哪一种代币分配（BTC和/或ETH）

- 决定赏金奖励的预算总额
- 从预算来看，确定三级奖励： - 你愿意发放的最小奖励 - 通常可发放的最高奖励 - 设置额外的限额以避免非常严重的漏洞被发现
- 确定赏金发放给谁（3是一个典型）
- 核心开发人员应该是赏金评委之一
- 当收到错误报告时，核心开发人员应该评估bug的严重性
- 在这个阶段的工作应该在私有仓库进行，并且在Github上的issue板块提出问题
- 如果这个bug需要被修复，开发人员应该在私有仓库编写测试用例来复现这个bug
- 开发人员需要修复bug并编写额外测试代码进行测试确保所有测试都通过
- 展示赏金猎人的修复；并将修复合并回公共仓库也是一种方式
- 确定赏金猎人是否有任何关于修复的其他反馈
- 赏金评委根据bug的*可能性*和*影响*来确定奖励的大小
- 在整个过程中保持赏金猎人参与讨论，并确保赏金发放不会延迟

有关三级奖励的例子，参见 Ethereum's Bounty Program：

> 奖励的价值将根据影响的严重程度而变化。 奖励轻微的“无害”错误从0.05 BTC开始。 主要错误，例如导致协商一致的问题，将获得最多5个BTC的奖励。 在非常严重的漏洞的情况下，更高的奖励是可能的（高达25 BTC）。

安全相关的文件和程序

当发布涉及大量资金或重要任务的合约时，必须包含适当的文档。有关安全性的文档包括：

规范和发布计划

- 规格说明文档，图表，状态机，模型和其他文档，帮助审核人员和社区了解系统打算做什么。
- 许多bug从规格中就能找到，而且它们的修复成本最低。
- 发布计划所涉及到的参考这里列出的详细信息和完成日期。

状态

- 当前代码被部署到哪里
- 编译器版本，使用的标志以及用于验证部署的字节码的步骤与源代码匹配
- 将用于不同阶段的编译器版本和标志
- 部署代码的当前状态（包括未决问题，性能统计信息等）

已知问题

- 合约的主要风险
  - 例如， 你可能会丢掉所有的钱，黑客可能会通过投票支持某些结果
- 所有已知的错误/限制
- 潜在的攻击和解决办法
- 潜在的利益冲突（例如，筹集的Ether将纳入自己的腰包，像Slock.it与DAO一样）

历史记录

- 测试（包括使用统计，发现的错误，测试时间）
- 已审核代码的人员（及其关键反馈）

程序

- 发现错误的行动计划（例如紧急情况选项，公众通知程序等）
- 如果出现问题，就可以降级程序（例如，资金拥有者在被攻击之前的剩余资金占现在剩余资金的比例）
- 负责任的披露政策（例如，在哪里报告发现的bug，任何bug赏金计划的规则）
- 在失败的情况下的追索权（例如，保险，罚款基金，无追索权）

联系信息

- 发现问题后和谁联系
- 程序员姓名和/或其他重要参与方的名称
- 可以询问问题的论坛/聊天室

安全工具

- Oyente - 根据这篇文章分析Ethereum代码以找到常见的漏洞。
- solidity-coverage - Solidity代码覆盖率测试
- Solgraph - 生成一个DOT图，显示了Solidity合约的功能控制流程，并highlight了潜在的安全漏洞。

Linters

Linters通过约束代码风格和排版来提高代码质量，使代码更容易阅读和查看。

- Solium - 另一种Solidity linting。
- Solint - 帮助你实施代码一致性约定来避免你合约中的错误的Solidity linting
- Solcheck - 用JS写的Solidity linter，（实现上）深受eslint的影响。

将来的改进

- **编辑器安全警告**：编辑器将很快能够实现醒常见的安全错误，而不仅仅是编译错误。 Solidity浏览器即将推出这些功能。
- **新的能够被编译成EVM字节码的函数式编程语言**： 像Solidity这种函数式编程语言相比面向过程编程语言能够保证功能的不变性和编译时间检查。通过确定性行为来减少出现错误的风险。（更多相关信息请参阅 这里, Curry-Howard 一致性和线性逻辑）

智能合约安全参考书目

很多包含代码，示例和见解的文档已经由社区编写完成。这里是其中的一些，你可以随意添加更多新的内容。

来自以太坊核心开发人员

- How to Write Safe Smart Contracts (Christian Reitwiessner)
- Smart Contract Security (Christian Reitwiessner)
- Thinking about Smart Contract Security (Vitalik Buterin)
- Solidity
- Solidity Security Considerations

来自社区

- http://forum.ethereum.org/discussion/1317/reentrant-contracts
- http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/
- http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
- http://hackingdistributed.com/2016/06/22/smart-contract-escape-hatches/
- http://martin.swende.se/blog/Devcon1-and-contract-security.html
- http://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf
- http://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour
- http://vessenes.com/ethereum-griefing-wallets-send-w-throw-considered-harmful
- http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal
- https://blog.blockstack.org/simple-contracts-are-better-contracts-what-we-can-learn-from-the-dao-6293214bad3a
- https://blog.slock.it/deja-vu-dao-smart-contracts-audit-results-d26bc088e32e
- https://blog.vdice.io/wp-content/uploads/2016/11/vsliceaudit_v1.3.pdf
- https://eprint.iacr.org/2016/1007.pdf
- https://github.com/Bunjin/Rouleth/blob/master/Security.md
- https://github.com/LeastAuthority/ethereum-analyses
- https://medium.com/@ConsenSys/assert-guards-towards-automated-code-bounties-safe-smart-contract-coding-on-ethereum-8e74364b795c
- https://medium.com/@coriacetic/in-bits-we-trust-4e464b418f0b
- https://medium.com/@hrishiolickel/why-smart-contracts-fail-undiscovered-bugs-and-what-we-can-do-about-them-119aa2843007
- https://medium.com/@peterborah/we-need-fault-tolerant-smart-contracts-ec1b56596dbc

- https://medium.com/zeppelin-blog/zeppelin-framework-proposal-and-development-roadmap-fdfa9a3a32ab
- https://pdaian.com/blog/chasing-the-dao-attackers-wake
- http://www.comp.nus.edu.sg/~loiluu/papers/oyente.pdf

Reviewers

The following people have reviewed this document (date and commit they reviewed in parentheses): Bill Gleim (07/29/2016 3495fb5) Bill Gleim (03/15/2017 0244f4e)

License