CSA GCR cloud security alliance
GREATER CHINA REGION ®

智能合约
安全指南

致谢

云安全联盟大中华区（简称：CSA GCR）区块链安全工作组在 2020 年 2 月份成立。由黄连金担任工作组组长，9 位领军人分别担任 9 个项目小组组长，分别有：知道创宇创始人&CEO 赵伟领衔数字钱包安全小组，北大信息科学技术学院区块链研究中心主任陈钟领衔共识算法安全小组，赛博英杰创始人&董事长谭晓生领衔交易所安全小组，安比实验室创始人郭宇领衔智能合约安全小组，世界银行首席信息安全架构师张志军领衔 Dapp 安全小组，元界 DNA 创始人兼 CEO 初夏虎领衔去中心化数字身份安全小组，北理工教授祝烈煌领衔网络层安全小组，武汉大学教授陈晶领衔数据层安全小组，零时科技 CEO 邓永凯领衔 AML 技术与安全小组。

区块链安全工作组现有 100 多位安全专家们，分别来自中国电子学会、耶鲁大学、北京大学、北京理工大学、世界银行、中国金融认证中心、华为、腾讯、知道创宇、慢雾科技、启明星辰、天融信、联想、OPPO、零时科技、普华永道、安永、阿斯利康等六十多家单位。

本白皮书主要由智能合约安全小组专家撰写，感谢以下专家的贡献：

原创作者：郭宇、黄连金、吴潇、姚昌林、赵伟（按照字母排序）

审核专家：郭文生、叶振强、于晓航、余弦、祝烈煌（按照字母排序）

贡献单位：长亭科技、知道创宇（按照字母排序）

关于研究工作组的更多介绍，请在 CSA 大中华区官网（https://c-csa.cn/research/）上查看。

如本白皮书有不妥当之处，敬请读者联系 CSA GCR 秘书处给与雅正！联系邮箱：info@c-csa.cn；云安全联盟 CSA 公众号：

序言

智能合约作为区块链的重要技术极大地扩展了区块链的应用场景与现实意义，被广泛地应用于金融、游戏、保险、物联网等多个领域，但同时也面临着严重的安全风险。相比于普通程序而言，智能合约的安全性不仅影响合约参与多方的公平性，还影响合约所管理的庞大数字资产的安全性。因此，对智能合约的安全性及相关安全漏洞开展研究显得尤为重要。

CSA GCR 对于智能合约的安全进行系统化研究，从不同的角度去分析智能合约的安全，从智能合约开发者角度，在第一章从智能合约的安全框架出发提出了测试标准，可以为开发者在内部安全测试中参考。第二章结合部分典型案例，具体剖析曾发生过的经典智能合约安全事件和深层次原因，有利于开发者了解已经发生过的智能合约安全事件，引以为戒。第三章分析总结智能合约最佳实践，包括针对智能合约生态常见问题的基本对策、社区最佳实践和安全开发资料，以及推荐工具，可以在开发过程中使用。第四章从第三方审计的角度介绍智能合约安全审计的 Checklist，供第三方审计参考使用。第五章做了简要的总结。

本文档主要用 Solidity 语言进行说明，因为绝大多数的智能合约是用 Solidity 进行编程的，但是大部分安全指南内容也可以用到其他编程语言开发的智能合约。智能合约的技术和编程语言在不停地发展，安全方面也需要与时俱进。本文档作为第一个版本抛砖引玉，希望专家们批评指正。

李雨航 Yale Li

CSA 大中华区主席兼研究院院长

目录

1. 测试标准

智能合约是部署和运行在区块链上的程序。借助区块链，智能合约可以实现资产管理、多方游戏等各类去中心化应用（dApp）。与传统程序一样，智能合约中存在漏洞也在所难免。然而不同的是，智能合约运行在更为开放的环境中，并天生带有金融属性，且升级成本极高。这意味着其对安全的要求更高，任何一点瑕疵都可能带来无法预估的后果。

智能合约安全本身是一个系统性和专业性极强的工程，需要综合考虑合约平台底层安全、合约设计与实现安全、合约生态工具安全、合约交互与数据安全等各方面，任何一个环节出现细微问题都会留下极大的隐患。本标准从以上各个方面分别阐述，并给出详细具体要求，旨在通过规范测试标准提升智能合约安全水平。

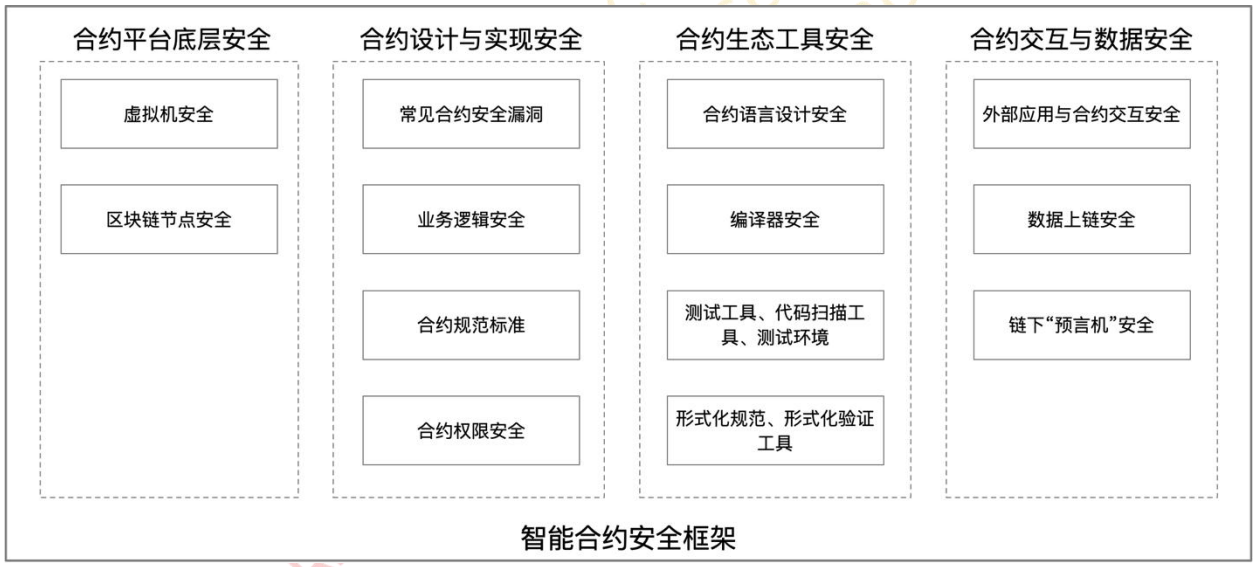

图 1-1

下面， 我们从合约平台底层安全、合约设计与实现安全、合约生态工具安全、合约交互与数据安全四个方面规范智能合约安全测试标准。

1.1 合约平台底层安全

本部分讨论智能合约平台相关的底层安全，包括虚拟机、区块链节点等。

1.1.1 虚拟机安全

虚拟机是智能合约运行所需的环境（例如应用最广泛的以太坊 EVM），关系着合约的执行结果，以及能否对区块链状态进行正确更改。虚拟机安全是智能合约安全的重要基石，合约执行功能的正确性、区块链系统的安全性都与之密切关联。智能合约必须能在虚拟机内正确且安全地执行。

具体要求：

1、不同操作符资源定价合理（如以太坊中对不同指令收取不同 Gas 费用）

2、准确执行操作符，操作符无二义性

3、能保证合约程序的终止性

4、能够安全处理异常调用

5、能保证合约对数据段的正确读取权限

6、不包含存在明显安全隐患的特性

7、不存在虚拟机逃逸漏洞

8、不存在任意代码执行漏洞

9、不存在 DOS 漏洞

10、预编译合约（内建合约、系统合约）调用与执行安全

1.1.2 区块链节点安全

完整的区块链系统由众多部署运行在不同机器上的节点构成。智能合约通过交易的形式在节点上部署和执行，而节点则会开放接口供外部调用或查询合约。节点需做好最坏情况下的安全防护，确保主机安全。智能合约的部署与运行应不影响区块链节点的正常工作，且不影响节点间的同步与共识。

具体要求：

1、区块链系统（节点）运营者需要做好安全防护，保障主机安全

2、区块链系统（节点）需要防止已部署合约代码被任意篡改

3、区块链系统（节点）需要防止针对已部署合约的非法调用

4、提供合约查询和调用服务的节点需确保自身状态正确且保持最新

5、防止合约部署或调用占用节点过多计算资源

6、防止合约部署或调用引起节点崩溃

7、能够正常防范针对节点发起的其他形式的攻击

1.2 合约设计与实现安全

本部分讨论智能合约代码实现层面的安全，包括常见安全漏洞、业务逻辑安全、规范标准、权限安全等。

1.2.1 常见合约安全漏洞

智能合约开发者应当熟悉各类常见的安全漏洞，在编写智能合约时注意规避。另外还尤其要关注传统程序与智能合约开发之间的差异。合约代码实现应规避常见安全漏洞。

具体要求：

1. 正常通过编译且无警告输出
2. 合约各功能能够正常执行
3. 不存在冗余代码
4. 能够安全进行和接受转账
5. 无「数值溢出」漏洞
6. 无「随机数预测」漏洞

7. 依赖的合约代码库安全可靠

8. 合理校验输入数据

9. 严格进行权限校验

10. 不存在未知步数循环

11. 无「重入（Reentrancy）」漏洞

12. 充分检查合约中硬编码的地址

13. 通知事件（Event）与代码实际执行保持一致

14. 关键逻辑判断不依赖可被矿工操纵的变量（区块哈希、时间戳等）

15. 关键逻辑判断不依赖可被用户任意操纵的变量（如合约内资产数量等）

16. 无严重消耗资源（如以太坊平台的 Gas、EOS 平台的 Ram）的操作

17. 合约正式部署上线前应该经过详细测试（核心代码测试覆盖率 95%以上）或第三方安全审计

1.2.2 业务逻辑安全

作为去中心化应用的核心部分，智能合约还需要可靠的设计。糟糕的设计可能会引入难以发现的深层次安全问题，这类问题往往无法简单通过扫描源代码发现。智能合约设计者需要综合考虑业务逻辑、多角色权限和博弈、区块链共识等多方面因素。合约业务逻辑设计与实现应不存在明显安全问题。

具体要求：

1. 合约的机制设计无明显缺陷

2. 合约代码实现与业务逻辑设计相符合，不存在歧义

3. 多角色的合约，各角色权限和优先级划分明确

4. 多角色的合约，各角色之前的公平性有一定保障

5. 存在博弈环节的合约，博弈过程无明显漏洞

6. 不存在「抢先交易」或「交易顺序依赖」的风险

7. 不存在「拒绝服务」（Denial-of-Service, DoS）风险

8. 不会因「薅羊毛」等行为给合约正常运行造成不良影响

9. 不存在交易参数签名重放问题

10. 业务逻辑不会与合约所属区块链系统共识机制产生冲突

1.2.3 合约规范标准

智能合约开发应当做到规范，遵循社区公认的代码最佳实践，以降低代码安全审计复杂度。智能合约本身表达比较灵活，而社区或特定行业会因为发展需要，对同类型的合约进行接口规范约定。这可以降低与各个合约的对接成本，并利于统一收录和交互。相反，如果合约规范标准得不到应有的重视，则会出现各类兼容性问题甚至安全隐患。特定类型合约应符合相应标准。

具体要求：

1. 函数及变量标明可见性，遵循变量声明和作用域的规范

2. 变量标明数据类型

3. 不使用已过期或即将废弃的用法

4. 代码风格简洁，命名规范

5. 特定类型合约实现严格按照相关合约标准（如 ERC20、ERC721），保持接口和实现的兼容性

6. 使用经过多方充分讨论和审计的合约标准和代码实现

7. 优先使用经过详细测试或审计的可靠合约代码库，并确保在理解的情况下进行使用

8. 规范使用内置函数

1.2.4 合约权限安全

部分智能合约会受管理员控制，而部分管理员甚至拥有极高的特殊权限，特定场景下会威胁合约其他用户的资产安全。为了智能合约安全，我们一方面需要警惕超级管理员作恶，另一方面需要考虑管理员身份被盗用所带来的后果。接口调用权限、管理员权限等需要划分清晰。

具体要求：

1. 限制管理员的特殊权限，防止权限过大，不可危及普通用户安全

2. 合约管理员账户应该通过多重签名钱包（合约）的方式来安全管理

3. 安全保护管理员私钥，防止因管理员身份被盗用对合约造成致命影响

4. 无其他极易发生「单点失效」的薄弱环节，必要时引入多签账户或 DAO 降低风险

1.3 合约生态工具安全

本部分讨论智能合约生态基础设施工具安全，包括合约语言、编译器、测试工具、代码扫描工具等。

1.3.1 合约语言设计安全

开发者会使用特定的智能合约语言编写智能合约，而合约语言本身的设计会从底层影响开发难度、开发效率以及合约安全性。

具体要求：

1. 具备完整明确的语义定义

2. 不存在容易导致漏洞的非确定语义

3. 语言开发文档描述完整，文档定义与语言实现保持一致

4. 不包含存在安全隐患的特性

1.3.2 编译器安全

编译器通过语法、语义分析，将开发者编写的源代码经过解析最终转换为目标代码（通常是机器码）。安全可信的编译器对于智能合约安全尤为重要：一方面，编译器需确保源代码到目标代码的正确翻译，不引入错误；另一方面，编译器还可具备一定的安全检测特性，从而可尽早帮助开发者发现源代码中的常见安全风险。

具体要求：

1. 能够正确地将源代码翻译成目标代码

2. 能够检查常见合约漏洞并给出清晰的提示

3. 编译优化不应引入新的代码漏洞或改变代码原始功能

4. 编译器的实现具备完整的测试框架和测试用例

1.3.3 测试工具、代码扫描工具、测试环境

智能合约平台应该为开发者提供良好的调试工具、代码扫描工具和测试环境，降低开发者测试成本，进而提高代码质量。

具体要求：

1. 备有完善的调试工具，方便开发者部署和调用合约，追踪状态变化

2. 备有先进的代码扫描工具，如利用静态代码扫描、动态代码分析、模糊测试等技术，供开发者快速扫描代码，对高风险处给出警示

3. 备有公开的测试网络，能够准确模拟正式主网

4. 备有高度自动化的测试框架，方面开发者进行单元测试、回归测试等操作，并提供准确的测试用例覆盖率分析报告

5. 支持连接正式主网，便于对已部署程序进行检查测试

1.3.4 合约形式化规范、形式化验证工具

采用形式化验证工具对合约进行可靠性验证、采用形式化语言为智能合约编写规范。

具体要求：

1. 开发者应该优先选择经过完整形式化证明的模版代码

2. 高价值的智能合约应考虑引入形式化验证技术来保障安全

3. 形式化验证工具能够对智能合约代码进行自动化的扫描和验证，提供详细的问题定位和修改建议

4. 形式化验证工具能够对通过检查的智能合约代码自动出具带有签名的验证报告，作为代码安全性的证明文件

1.4 合约交互与数据安全

本部分讨论智能合约与外部应用发生交互的安全性、链上数据安全以及链下“预言机”安全。

1.4.1 外部应用与合约交互安全

外部应用或用户与合约交互的形式包括部署、查询、调用。用户通常会使用特定工具对合约发起操作，合约则需提前公示准确的接口描述。用户不应该随意调用未知的合约，合约也需要对用户的输入做严格检查。

具体要求：

1. 外部应用或用户可正常通过接口访问或调用智能合约功能

2. 合约需要为外部调用者提供详细的接口说明

3. 外部应用或用户应使用安全的工具与合约交互

4. 合约各个函数实现需对传入参数做严格校验

5. 检查合约代码实现与触发的外部事件是否一致，事件是否会非预期地触发

6. 需评估不同种类的合约经过组合使用后的安全风险

7. 不调用未知合约

8. 对接外部合约之前需要有兼容性的安全评估和测试

9. 调用外部合约需要对其返回值等进行校验

1.4.2 数据上链安全

真实业务数据上链需注意数据安全问题。

具体要求：

1. 防范敏感或隐私数据的非预期泄漏

2. 防范敏感或隐私数据被非授权使用或访问

3. 需解决特定应用场景下的数据可靠性和真实性问题

4. 需考虑特定数据上链后公开且不可更改与法律法规的合规性问题

1.4.3 链下“预言机”安全

智能合约通常无法直接获取外部世界的真实信息，因此需要预言机（Oracle）为其提供安全获取外界信息的能力，预言机是链接智能合约与外部世界信息的基础设施，应用场景广泛。

具体要求：

1. 根据具体业务选择合适的预言机，比如常规合约可选择中心化预言机，而高额资产交易合约应优先选择去中心化预言机。

2. 应确保预言机所提供的数据不被轻易恶意操纵和篡改

3. 应确保预言机能保障数据真实性、时效性和可用性

4. 合约实现逻辑中应对所预言机可能发生的所有异常进行检测和处理，以保障各类情况下合约自身的安全

2. 典型案例

智能合约开发者很容易在缺乏经验和警惕的情况下，写出存在安全漏洞的智能合约代码，并且未经充分的测试和审计，匆忙部署上线，最终酿成严重的安全事件。通过分析经典的智能合约安全事件案例，可以更深入地理解智能合约安全。

2.1 重入漏洞

2016 年 6 月 18 日，针对 DAO 合约的攻击导致超过 3,600,000 个 Ether 的损失。攻击源于 DAO 一个关键合约引入了重入漏洞，黑客可以对转账函数进行递归重入，在不减少个人余额的情况下不断地提走以太币。这次攻击之后的以太坊硬分叉导致了以太坊社区的分裂。

2.2 整数溢出漏洞

整数溢出漏洞也是智能合约中常见的安全漏洞。例如， uint256 是以太坊智能合约语言 Solidity 中最常用的无符号整数类型，它可表示的整数范围为 0 ~ 2256 - 1。当程序中的计算结果超过此范围时，就会发生整数溢出。且一旦发生，后果通常十分严重。在智能合约中，整数通常与 Token 账户余额或者其他关键程序逻辑相关。利用整数溢出漏洞，精心构造传入参数，可以让程序极大地偏离本来的设计意图，从而引发一系列难以预知的问题。

2018 年 4 月 22 日，黑客攻击了美链 (BEC) 的 Token 合约，通过一个整数溢出安全漏洞，将大量 Token 砸向交易所，导致 BEC 的价格几乎归零。

2018 年 4 月 25 日，SMT 爆出类似的整数溢出漏洞，黑客通过漏洞制造和抛售了天文数字规模的 Token，导致 SMT 价格崩盘。

整数溢出漏洞可能造成以下严重后果：

1. 使程序偏离设计意图

2. 实现 Token 无限量增发

3. 绕过权限控制或余额校验

4. 完成一些虚假操作

5. 被 Token 管理员利用，留下作恶空间

2.3 合约库安全问题

2017 年 11 月 06 日，Parity 多重签名钱包漏洞导致了超过 513,701 个以太币被锁死，至今关于以太坊是否需要通过硬分叉方式升级 (EIP999) 争论还在继续。

2.4 复杂 DeFi 协议组合安全

2020 年 2 月 15 日、2 月 18 日，DeFi 项目 bZx 借贷协议遭遇两次攻击，先后损失 35 万美元和 64 万美元。两次攻击手法复杂且不完全相同，利用新型借贷产品“闪电贷”将攻击成本降至最低并扩大攻击收益，同时又与价格预言机、杠杆交易、借贷、价格操纵、抢先交易紧密关联，充分利用了多个 DeFi 产品间的可组合性来达到攻击目的。

2020 年 4 月 19 日，去中性化借贷协议 Lendf.Me 遭遇黑客攻击，合约内价值 2500 万美元的资产被洗劫一空，直接原因在于产品本身的可重入问题和特殊的 ERC-777 类型代币 imBTC 组合之后引入的新的安全风险。同时因为此问题被攻击的还有 Uniswap 去中心化交易所。

2.5 非标准接口

以最被广泛接受的智能合约 Token 标准 ERC20 为例，很多 Token 合约未参照 ERC20 标准实现，这给 dApp 开发带来很大的困扰。数以千计的已部署 Token 合约曾经参考了以太坊官网（现已修复）以及 OpenZeppelin（52120a8c42 [2017 年 3 月 21 日] ~ 6331dd125d [2017 年 7 月 13 日]) 给出的不规范模版代码，多个函数实现没有遵循 ERC20 规范，导致 Solidity 编译器升级至 0.4.22 后出现严重的兼容性问题，无法正常转账。

2.6 管理员权限过高或较中心化

2018 年 7 月 10 日，加密货币交易平台 Bancor 称遭到攻击，丢失了折算法币金额为 1250 万美金的以太坊，1000 万美金的 Bancor 代币和 100 万美金的 Pundix 代币。这次攻击黑客并非利用了 Bancor 平台合约的漏洞，而是通过盗取了其智能合约管理员账户的私钥。Bancor 主要合约包括 SmartToken 和 BancorConverter，分别为 ERC20 Token 合约以及与业务相关的 Token 转换交易合约。此次被盗事件与 BancorConverter 合约有关，攻击者极有可能获取了合约管理员账户的私钥，对转换代币合约 BancorConverter 拥有极高权限。owner 作为该合约的所有者和管理员，有唯一的权限通过 withdrawTokens() 方法提走合约中的全部 ERC20 Token 至任意地址。事件发生后不少人质疑 Bancor 项目智能合约中 owner 管理员的超级权限，甚至称之为“后门”。

2.7 业务逻辑与区块链共识产生冲突

曾经红极一时的类 Fomo3D 游戏，就曾因「随机数预测」攻击和「阻塞交易」攻击丧失游戏公平性，沦为黑客淘金的重灾地。

类 Fomo3D 游戏的空投机制利用「随机数」来控制中奖概率，但是由于随机数的来源都是区块或者交易中特定的一些公开参数，如交易发起者地址、区块时间戳、区块难度等，因此在以太坊上可以很容易的预测所谓的「随机数」。

而「阻塞交易」攻击是黑客通过高额手续费吸引矿工优先打包，并利用合约自动判断游戏进行状态，以之作为是否采取攻击的依据。黑客最终能够以较低成本堵塞区块，每个区块中仅打包很少的交易（降低他人交易被打包的可能性），使得游戏快速结束，并提高自己获得最终大奖的概率。所有采用类似机制，即需要玩家抢在某个时间范围内完成某种竞争操作的智能合约，都会受此威胁。

类 Fomo3D 游戏中黑客利用了以太坊共识协议的特点，找到游戏机制的漏洞进而发起攻击。智能合约游戏或应用的业务设计是一项极其复杂的系统工程，除了代码本身，还涉及到平台外部环境，博弈论等方面问题，这也是容易被黑客攻击的环节。因此，在业务设计过程中应尤为慎重，必须充分考虑区块链的特性。

2.8 开发者应持续关注更多安全案例

除了上面提到的典型案例外，智能合约领域还有很多值得研究的漏洞案例，并且各类新型的攻击手法层出不穷。智能合约本身还很不完善，并且还处在爆发式地发展中，存在着各类未知风险。 为了开发成熟、安全、可靠的智能合约，开发者应持续关注更多安全案例，从中学习经验教训。

3. 最佳实践

从智能合约开发者方面来看，区块链项目的爆发吸引了大量开发者，而这些开发者往往只能从简单的参考和修改网上良莠不齐甚至有严重漏洞的示例代码开始进行学习开发。对于资深智能合约开发者而言，实现安全无漏洞的智能合约依旧是不小的挑战。事实上，开发者可以通过遵循最佳实践的方法，提高代码质量，降低安全风险。

3.1 基本对策

针对智能合约安全生态中常见的问题，有如下几点基本对策：

1. 规范合约安全开发和发布流程
2. 引入形式化方法保障合约安全
3. 开发更优秀的合约语言、编译器和虚拟机
4. 社区应重视合约标准

对策一：规范合约安全开发和发布流程

目前鲜有智能合约开发团队具备完善的合约开发和发布流程。我们建议开发团队按照从设计、开发、测试、审计到部署和监控的顺序，安全规范开发。尤其需要重视设计、测试以及必要时引入外部第三方安全审计。

开发者可阅读智能合约安全开发资料，熟悉常见安全漏洞和安全事故，并自觉规避。

此外，开发者也可在开发过程中引入开源的智能合约安全自动扫描工具，从而规范

合约代码书写，并提前发现常见的安全风险。这些工具通常采用静态分析、符号执行、污点分析、模糊测试等技术，可较快扫描出常见的漏洞和高风险代码。

对策二：引入形式化方法保障合约安全

「形式化方法」特指用来对软件和硬件系统进行规范，设计和验证的数学理论及相关工具。

其中，形式化规范 (formal specification) 通过数学语言对系统的预期行为 (例如将数量 S 的 token 从账户 A 转移到账户 B) 和性质 (例如转账不会造成账户 B 额度的溢出) 进行严格和全面的定义。形式化规范往往定义了系统的正确性和安全性。

给定一个系统的形式化规范，我们即可以从规范出发开始迭代设计和实现出这个系统。在迭代的每一步中，我们可以通过精化 (refinement)、集成 (synthesis) 、形式化证明在内的一系列方法在数学上严格的保证迭代产生的系统和迭代前的规范或者系统保持一致。

除了从形式化规范出发设计和实现一个系统，我们也可以使用包括符号执行 (symbolic execution)、模型检测 (model checking)和形式化证明 (formal proving) 在内的一系列方法验证已有的设计和实现与该规范保持一致。所有的定义都是通过严格的数学语言描述，推导和检查也是严格的数学推导和证明。在实践中，推导和证明无法进行下去往往意味着设计和实现中存在不符合规范的 bug。通过分析推导和证明卡壳的位置和原因，往往可以定位出 bug 在设计和实现中具体位置和成因。

通过形式化方法对智能合约的行为进行精确地数学推理和证明，可以保证智能合约的行为符合开发者的设计意图，从而保证代码的安全性。基于数学证明的形式化方法为开发高可信的软件系统提供了有效的手段。

对策三：开发更优秀的合约语言、编译器和虚拟机

部分智能合约代码安全问题实际可以通过改进智能合约语言、编译器和虚拟机来规避。而事实上不少漏洞也正是因为语言设计、编译器或虚拟机实现问题而产生。

很多智能合约语言一方面提供高度的灵活性和强大的表达能力，另一方面又缺乏完

备的开发文档和语义定义，特别是对语言版本之间不兼容性的醒目提示，所有这些常常会导致开发者意外地忽略语言中的重要细节和混淆语言中的各种概念，例如以太坊智能合约语言 Solidity 中忽略地址对齐问题导致短地址攻击，混淆 tx.origin 和 msg.sender 导致的身份认证错误，blockhash 的确定性导致的其不能作为随机数种子，Solidity 不同版本对返回值的不同校验方式导致的旧智能合约不能正常调用等。目前流行的智能合约语言，由于历史遗留问题，都或多或少存在着一些设计上的失误，进而容易误导开发者写出高风险的代码。社区完全可以尝试重新设计和拥抱更好、更安全的智能合约语言；甚至可以发展出面向特定领域的合约语言，在减少灵活性和表达能力的同时降低开发者出错的可能。

对策四：社区应重视合约标准

区块链社区的特点是组织较松散，这也间接影响了智能合约标准的质量以及开发者对标准提案的重视程度。

目前以太坊合约标准提案由志愿者自行提出和发表，由社区成员自由讨论，现已发展出包括 ERC-20、ERC-721、ERC-998、ERC-1155、ERC-1178 在内的一系列合约提案或标准。

然而，存在两个隐患可能会威胁合约安全。一方面，标准提案或代码实现质量良莠不齐。前文提到曾发生合约模版代码存在问题和有安全风险的提案实现进入代码仓库。这种问题一旦发生，影响都十分严重。社区必须重视提案和代码实现质量。另一方面，合约标准提案仅通过文档的形式描述，难以检查和验证代码是否符合标准。开发者根据提案实现代码时，不能方便地快速测试正确性，从而容易引发兼容性和安全性问题。ERC-721 在这方面有不错的表现，社区提供了基于大量测试用例的代码验证器。

事实上，社区也可以通过为合约标准定义「形式化规范」，进一步严格定义合约实现的各种预期行为，并可以与前文提到的形式化方法实现对接。

3.2 社区最佳实践和安全开发资料

开发者可以参考和遵循以下社区最佳实践和安全开发资料。

- DASP Top 10
- Ethereum Smart Contract Best Practices (ConsenSys Diligence)
- Ethereum Smart Contracts Security CheckList From Knownsec 404 Team
- A guide to EOS smart contract security best practices (Slowmist)
- Solidity Security Blog

3.3 推荐工具

智能合约安全生态已经产生了一些优秀的代码安全工具和分析工具，研究人员可以借助这些工具进行智能合约安全分析，开发者也可以在开发测试过程中引入这些工具，以提升代码质量，提前发现安全风险。

3.3.1 安全工具

Slither

Slither 是一个 Solidity 智能合约静态分析框架，还提供了一个 API 来给研究人员通过自定义脚本审计 Solidity 代码。我们可以使用 Slither 做到：

1. 识别能够修改变量值的代码
2. 隔离受特定变量值影响的条件逻辑语句
3. 查找能够调用特定函数的其他函数

项目地址：https://github.com/crytic/slither

Mythril

Mythril 是一款针对 EVM 字节码进行安全审计的工具，同时支持在线合约审计。

项目地址：https://github.com/ConsenSys/mythril

Echidna

Echidna 是一款针对 EVM 字节码进行安全审计的工具，利用了模糊测试技术，同时支持和 truffle 集成使用。

项目地址：https://github.com/crytic/echidna

Ethersplay

Ethersplay 是用于 Binary Ninja 的插件，可以用于分析 EVM 字节码，并用图形化的方式呈现函数调用流程。

项目地址：https://github.com/crytic/ethersplay

3.3.2 分析工具

Beosin-VaaS

Beosin-VaaS“一键式”智能合约自动形式化验证工具，可精确定位到有风险的代码位置并指出风险原因，有效的检测智能合约常规安全漏洞、安全属性和功能正确性，精确度高达 95%以上，为智能合约代码提供“军事级”的安全验证。

官网地址：https://beosin.com/vaas/index.html

Solgraph

Solgraph 用于生成智能合约函数间调用关系的图，方便快速了解智能合约函数间的调用关系。

项目地址：https://github.com/raineorshine/solgraph

Sol2uml

Sol2uml 用于以 UML 图的形式生成智能合约函数之间的调用关系。

项目地址：https://github.com/naddison36/sol2uml

4：以太坊智能合约审计 CheckList

下面是知道创宇 404 区块链安全研究团队提供的以太坊智能合约审计 CheckList。

知道创宇 404 区块链安全研究团队参考并整理兼容了各大区块链安全研究团队的研究成果，把以太坊合约审计中遇到的问题分为 5 大种，包括编码规范问题、设计缺陷问题、编码安全问题、编码设计问题、编码问题隐患。其中涵盖了超过 29 种会出现以太坊智能合约审计过程中遇到的问题。帮助智能合约的开发者和安全工作者快速入门智能合约安全。

4.1 编码规范问题

(1) 编译器版本

合约代码中，应指定编译器版本。建议使用最新的编译器版本

pragma solidity ^0.4.25;

老版本的编译器可能会导致各种已知的安全问题，例如 https://paper.seebug.org/631/#44-dividenddistributor

v0.4.23 更新了一个编译器漏洞，在这个版本中如果同时使用了两种构造函数，即

```
contract a {
    function a() public{
        ...
    }
    constructor() public{
        ...
    }
}
```

会忽略其中的一个构造函数，该问题只影响 v0.4.22

v0.4.25 修复了下面提到的未初始化存储指针问题。

https://etherscan.io/solcbuginfo

(2) 构造函数书写问题

对应不同编译器版本应使用正确的构造函数，否则可能导致合约所有者变更

在小于 0.4.22 版本的 solidify 编译器语法要求中，合约构造函数必须和合约名字相等， 名字受到大小写影响。如：

```
contract Owned {
    function Owned() public{
    }
```

在 0.4.22 版本以后，引入了 constructor 关键字作为构造函数声明，但不需要 function

```
contract Owned {
    constructor() public {
    }
```

如果没有按照对应的写法，构造函数就会被编译成一个普通函数，可以被任意人调用，会导致 owner 权限被窃取等更严重的后果。

(3) 返回标准

遵循 ERC20 规范，要求 transfer、approve 函数应返回 bool 值，需要添加返回值代码

```
function transfer(address _to, uint256 _value) public returns (bool success)
```

而 transferFrom 返回结果应该和 transfer 返回结果一致。

(4) 事件标准

遵循 ERC20 规范，要求 transfer、approve 函数触发相应的事件

```
function approve(address _spender, uint256 _value) public returns (bool success){
```

```
        allowance[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value)
        return true
```

(5) 假充值问题

转账函数中，对余额以及转账金额的判断，需要使用 require 函数抛出错误，否则会导致外部程序错误地判断为转账成功。

```
function transfer(address _to, uint256 _value) returns (bool success) {
        if (balances[msg.sender] >= _value && _value > 0) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
}
```

上述代码可能会导致假充值。

正确代码如下：

```
function transfer(address _to, uint256 _amount)    public returns (bool success) {
        require(_to != address(0));
        require(_amount <= balances[msg.sender]);
        balances[msg.sender] = balances[msg.sender].sub(_amount);
        balances[_to] = balances[_to].add(_amount);
```

```
        emit Transfer(msg.sender, _to, _amount);

        return true;

    }
```

4.2 设计缺陷问题

(1) approve 授权函数条件竞争

approve 函数中应避免条件竞争。在修改 allowance 前，应先修改为 0，再修改为 _value。

这个漏洞的起因是由于底层矿工协议中为了鼓励矿工挖矿，矿工可以自己决定打包什么交易，为了收益更大，矿工一般会选择打包 gas price 更大的交易，而不会依赖交易顺序的前后。

通过置 0 的方式，可以在一定程度上缓解条件竞争中产生的危害，合约管理人可以通过检查日志来判断是否有条件竞争情况的发生，这种修复方式更大的意义在于，提醒使用 approve 函数的用户，该函数的操作在一定程度上是不可逆的。

```
function approve(address _spender, uint256 _value) public returns (bool success){

        allowance[msg.sender][_spender] = _value;

        return true
```

上述代码就有可能导致条件竞争。

应在 approve 中加入

```
require((_value == 0) || (allowance[msg.sender][_spender] == 0));
```

将 allowance 先改为 0 再改为对应数字，也可以使用 increaseApprove 和 decreaseApprove 函数来更改授权值

(2) 循环 Dos 问题

[1] 循环消耗问题

在合约中，不推荐使用太多次的循环

在以太坊中，每一笔交易都会消耗一定量的 gas，而实际消耗量是由交易的复杂度决定的，循环次数越大，交易的复杂度越高，当超过允许的最大 gas 消耗量时，会导致交易失败。

真实世界事件

Simoleon (SIM)

https://paper.seebug.org/646/

Pandemica

https://bcsec.org/index/detail/id/260/tag/2

[2] 循环安全问题

合约中，应尽量避免循环次数受到用户控制，攻击者可能会使用过大的循环来完成 Dos 攻击

当用户需要同时向多个账户转账，我们需要对目标账户列表遍历转账，就有可能导致 Dos 攻击。

```
function Distribute(address[] _addresses, uint256[] _values) payable returns(bool){
    for (uint i = 0; i < _addresses.length; i++) {
        transfer(_addresses[i], _values[i]);
    }
    return true;
}
```

遇到上述情况是，推荐使用 withdrawFunds 来让用户取回自己的代币，而不是发送

给对应账户，可以在一定程序上减少危害。

上述代码如果控制函数调用，那么就可以构造巨大循环消耗 gas，造成 Dos 问题

(3) 冻结账户检测

当合约中存在账户冻结设计时，在转移代币的过程中应当校验代币来源账户、目标账户是否被冻结。

在一些智能合约中可能会存在冻结账户逻辑设计，例如常见的 freezen()函数，部分合约在进行代币转移时只判断了代币来源账户是否处于冻结状态而未判断代币的接受账户是否处于冻结状态，导致向冻结的账户转账后转移的资产无法再从中转出。

下述代码就存在冻结账户检测缺陷：

```
/* A contract attempts to get the coins */

function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {

        if (frozenAccount[_from]) throw;

        if (balanceOf[_from] < _value) throw;

        if (balanceOf[_to] + _value < balanceOf[_to]) throw;

        if (_value > allowance[_from][msg.sender]) throw;

        balanceOf[_from] -= _value;

        balanceOf[_to] += _value;

        allowance[_from][msg.sender] -= _value;

        Transfer(_from, _to, _value);

        return true;
}
```

应该在校验代币来源账户是否冻结的同时校验目标账户是否被冻结：

```
    /* A contract attempts to get the coins */
    function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {
        if (frozenAccount[_from]) throw;
        if (frozenAccount[_to]) throw;
        if (balanceOf[_from] < _value) throw;
        if (balanceOf[_to] + _value < balanceOf[_to]) throw;
        if (_value > allowance[_from][msg.sender]) throw;
        balanceOf[_from] -= _value;
        balanceOf[_to] += _value;
        allowance[_from][msg.sender] -= _value;
        Transfer(_from, _to, _value);
        return true;
    }
```

(4) Pausable 模块继承

建议主合约继承 Pausable Ownable ERC20 标准模块，当出现重大异常时可以暂停所有交易

```
``` {.solidity}
contract Ownable {
  address public owner;
```
```

```
    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
    constructor() public {
        owner = msg.sender;
    }
    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }
    function transferOwnership(address _newOwner) public onlyOwner {
        _transferOwnership(_newOwner);
    }
    function _transferOwnership(address _newOwner) internal {
        require(_newOwner != address(0));
        emit OwnershipTransferred(owner, _newOwner);
        owner = _newOwner;
    }
}
contract Pausable is Ownable {
    event Pause();
    event Unpause();
```

```
    bool public paused = false;
    modifier whenNotPaused() {
        require(!paused);
        _;
    }
    modifier whenPaused() {
        require(paused);
        _;
    }
    function pause() onlyOwner whenNotPaused public {
        paused = true;
        emit Pause();
    }
    function unpause() onlyOwner whenPaused public {
        paused = false;
        emit Unpause();
    }
}
contract StandardToken is Pausable {
    function transfer(address _to, uint256 _value) public whenNotPaused returns (bool) {
```

```
        //...
    }
    function transferFrom(address _from, address _to, uint256 _value) public
whenNotPaused returns (bool) {
        //...
    }
}
```

4.3 编码安全问题

(1) 溢出问题

[1] 算术溢出

在调用加减乘除时，应使用 safeMath 库来替代，否则容易导致算数上下溢，造成不可避免的损失

```
pragma solidity ^0.4.18;
contract Token {
    mapping(address => uint) balances;
    uint public totalSupply;
    function Token(uint _initialSupply) {
        balances[msg.sender] = totalSupply = _initialSupply;
    }
    function transfer(address _to, uint _value) public returns (bool) {
        require(balances[msg.sender] - _value >= 0); //可以通过下溢来绕过判断
```

```
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        return true;
    }

    function balanceOf(address _owner) public constant returns (uint balance) {
        return balances[_owner];
    }
}
```

balances[msg.sender] - _value >= 0 可以通过下溢来绕过判断。

通常的修复方式都是使用 openzeppelin-safeMath，但也可以通过对不同变量的判断来限制，但很难对乘法和指数做什么限制。

正确的写法如下：

```
function transfer(address _to, uint256 _amount)    public returns (bool success) {
        require(_to != address(0));
        require(_amount <= balances[msg.sender]);
        balances[msg.sender] = balances[msg.sender].sub(_amount);
        balances[_to] = balances[_to].add(_amount);
        emit Transfer(msg.sender, _to, _amount);
        return true;
}
```

真实世界事件

Hexagon

代币变泡沫，以太坊 Hexagon 溢出漏洞比狗庄还过分

SMT/BEC

Solidity 合约中的整数安全问题——SMT/BEC 合约整数溢出解析

[2] 铸币烧币溢出问题

铸币函数中，应对 totalSupply 设置上限，避免因为算术溢出等漏洞导致恶意铸币增发

```
function TokenERC20(
    uint256 initialSupply,
    string tokenName,
    string tokenSymbol
) public {
    totalSupply = initialSupply * 10 ** uint256(decimals);
    balanceOf[msg.sender] = totalSupply;
    name = tokenName;
    symbol = tokenSymbol;
}
```

上述代码中就未对 totalSupply 做限制，可能导致指数算数上溢。

正确写法如下：

```
contract OPL {
```

```
    // Public variables
    string public name;
    string public symbol;
    uint8 public decimals = 18; // 18 decimals
    bool public adminVer = false;
    address public owner;
    uint256 public totalSupply;
    function OPL() public {
        totalSupply = 210000000 * 10 ** uint256(decimals);
        ...
}
```

(2) 重入漏洞

智能合约中避免使用 call 来交易，避免重入漏洞

在智能合约中提供了 call、send、transfer 三种方式来交易以太坊，其中 call 最大的区别就是没有限制 gas，而其他两种在 gas 不够的情况下都会报 out of gas。

重入漏洞有几大特征。 1、使用了 call 函数作为转账函数 2、没有限制 call 函数的 gas 3、扣余额在转账之后 4、call 时加入了()来执行 fallback 函数

```
function withdraw(uint _amount) {
    require(balances[msg.sender] >= _amount);
    msg.sender.call.value(_amount)();
    balances[msg.sender] -= _amount;
```

```
}
```

上述代码就是一个简单的重入漏洞的 demo。通过重入注入转账，将大量合约代币递归转账而出。

对于可能存在的重入问题，尽可能的使用 transfer 函数完成转账，或者限制 call 执行的 gas，都可以有效的减少该问题的危害。

```
contract EtherStore {
    // initialise the mutex
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;
    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }
    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
```

```
            balances[msg.sender] -= _weiToWithdraw;
            lastWithdrawTime[msg.sender] = now;
            // set the reEntrancy mutex before the external call
            reEntrancyMutex = true;
            msg.sender.transfer(_weiToWithdraw);
            // release the mutex after the external call
            reEntrancyMutex = false;
        }
    }
```

上述代码是一种用互斥锁来避免递归防护方式。

真实事件:

The Dao

The DAO

The DAO address

(3) call 注入

call 函数调用时，应该做严格的权限控制，或直接写死 call 调用的函数

在 EVM 的设计中，如果 call 的参数 data 是 0xdeadbeef(假设的一个函数名) + 0x0000000000.....01，这样的话就是调用函数

call 注入可能导致代币窃取，权限绕过，通过 call 注入可以调用私有函数，甚至部分高权限函数。

```
addr.call(data);
```

addr.delegatecall(data);

addr.callcode(data);

如 delegatecall，在合约内必须调用其它合约时，可以使用关键字 library，这样可以确保合约是无状态而且不可自毁的。通过强制设置合约为无状态可以一定程度上缓解储存环境的复杂性，防止攻击者通过修改状态来攻击合约。

真实世界事件

call 注入

以太坊智能合约 call 注入攻击

以太坊 Solidity 合约 call 函数簇滥用导致的安全风险

(4) 权限控制

合约中不同函数应设置合理的权限

检查合约中各函数是否正确使用了 public、private 等关键词进行可见性修饰，检查合约是否正确定义并使用了 modifier 对关键函数进行访问限制，避免越权导致的问题。

```
function initContract() public {
    owner = msg.sender;
}
```

上述代码作为初始函数不应该为 public。

真实世界事件：

Parity Multi-sig bug 1

Parity Multi-sig bug 2

Rubixi

(5) 重放攻击

合约中如果涉及委托管理的需求，应注意验证的不可复用性，避免重放攻击

在资产管理体系中，常有委托管理的情况，委托人将资产给受托人管理，委托人支付一定的费用给受托人。这个业务场景在智能合约中也比较普遍。

这里举例子为 transferProxy 函数，该函数用于当 user1 转 token 给 user3，但没有 eth 来支付 gasprice，所以委托 user2 代理支付，通过调用 transferProxy 来完成。

```
function transferProxy(address _from, address _to, uint256 _value, uint256 _fee,
    uint8 _v, bytes32 _r, bytes32 _s) public returns (bool){
    if(balances[_from] < _fee + _value
        || _fee > _fee + _value) revert();
    uint256 nonce = nonces[_from];
    bytes32 h = keccak256(_from,_to,_value,_fee,nonce,address(this));
    if(_from != ecrecover(h,_v,_r,_s)) revert();
    if(balances[_to] + _value < balances[_to]
        || balances[msg.sender] + _fee < balances[msg.sender]) revert();
    balances[_to] += _value;
    emit Transfer(_from, _to, _value);
    balances[msg.sender] += _fee;
    emit Transfer(_from, msg.sender, _fee);
    balances[_from] -= _value + _fee;
    nonces[_from] = nonce + 1;
    return true;
```

}

这个函数的问题在于 nonce 值是可以预判的，其他变量不变的情况下，可以进行重放攻击，多次转账。

漏洞来自于 Defcon2018 演讲议题

Replay Attacks on Ethereum Smart Contracts Replay Attacks on Ethereum Smart Contracts pdf

(6) 一致性检查

智能合约中更新逻辑(例如：代币数量更新、授权转账额度更新等)在进行更新操作时往往都会伴随着对操作对象的检查逻辑(例如：防溢出检查、授权转账额度检查等)，而当更新对象与检查对象不一致时可导致检查操作无效，从而无视检查逻辑，执行意料之外的逻辑，这里以合约中的授权转账函数为例做简要说明：

function transferFrom(address _from, address _to, uint256 _value) returns (bool success)函数用于授权给他人进行代为转账，在转账时会对 allowance[_from][msg.sender]即授权转账额度进行检查，通过检查后在转账的同时会更新一次授权转账的额度，当更新逻辑中的更新对象与检查逻辑中的检查对象不一致时，被授权转账用户的授权转账额度将不会改变，导致被授权转账用户可以转走授权账户的所有资产，存在风险的代码如下所示：

```
/// @notice send `_value` token to `_to` from `_from` on the condition it is approved by `_from`

/// @param _from The address of the sender

/// @param _to The address of the recipient

/// @param _value The amount of token to be transferred

/// @return Whether the transfer was successful or not

function transferFrom(address _from, address _to, uint256 _value) public returns
```

```
(bool success) {
                require(balances[_from] >= _value);                 // Check if the sender
has enough
                require(balances[_to] + _value >= balances[_to]);  // Check for overflows
                require(_value <= allowed[_from][msg.sender]);     // Check allowance
              balances[_from] -= _value;
                balances[_to] += _value;
              allowed[_from][_to] -= _value;
                Transfer(_from, _to, _value);
                return true;
        }
```

在上述代码中下面这一行代码用于检测授权转账的额度：

```
require(_value <= allowed[_from][msg.sender]);
```

而下面这一行代码用于更新授权转账的额度：

```
allowed[_from][_to] -= _value;
```

可以看到判断逻辑与更新逻辑中的操作对象完全不一致，从而导致在设置转账额度权限后，被授权转账的用户能够持续转账，直到转完授权账户的所有余额为止。

正确的写法应该是保持判断逻辑与更新逻辑中的操作对象一致且为被授权转账用户的授权转账额度：

```
require(_value <= allowed[_from][msg.sender]);
……
allowed[_from][msg.sender]); -= _value;
```

4.4 编码设计问题

(1) 地址初始化问题

涉及到地址的函数中，建议加入 require(_to!=address(0))验证，有效避免用户误操作或未知错误导致的不必要的损失

由于 EVM 在编译合约代码时初始化的地址为 0，如果开发者在代码中初始化了某个 address 变量，但未赋予初值，或用户在发起某种操作时，误操作未赋予 address 变量值，但在下面的代码中操作了这个变量，就可能导致不必要的安全风险。

这样的检查可以以最简单的方式避免未知错误、短地址攻击等问题的发生。

(2) 判断函数问题

及到条件判断的地方，使用 require 函数而不是 assert 函数，因为 assert 会导致剩余的 gas 全部消耗掉，而他们在其他方面的表现都是一致的

值得注意的是，assert 存在强制一致性，对于固定变量的检查来说，assert 可以用于避免一些未知的问题，因为他会强制终止合约并使其无效化，在一些固定条件下，assert 更适用。

(3) 余额判断问题

不要假设合约创建时余额为 0，可以强制转账

谨慎编写用于检查账户余额的不变量，因为攻击者可以强制发送 wei 到任何账户，即使 fallback 函数 throw 也不行。

攻击者可以用 1wei 来创建合约，然后调用 selfdestruct(victimAddress)来销毁，这样余额就会强制转移给目标，而且目标合约没有代码执行，无法阻止。

值得注意的是，在打包过程中，攻击者可以通过条件竞争在合约创建前转账，这样在合约创建时余额就不为 0.

(4) 转账函数问题

在完成交易时，默认情况下推荐使用 transfer 而不是 send 完成交易

当 transfer 或者 send 函数的目标是合约时，会调用合约的 fallback 函数，但 fallback 函数执行失败时。

transfer 会抛出错误并自动回滚，而 send 会返回 false，所以在使用 send 时需要判断返回类型，否则可能会导致转账失败但余额减少的情况。

```
function withdraw(uint256 _amount) public {
    require(balances[msg.sender] >= _amount);
    balances[msg.sender] -= _amount;
    etherLeft -= _amount;
    msg.sender.send(_amount);
}
```

上面给出的代码中使用 send() 函数进行转账，因为这里没有验证 send() 返回值，如果 msg.sender 为合约账户 fallback() 调用失败，则 send() 返回 false，最终导致账户余额减少了，钱却没有拿到。

(5) 代码外部调用设计问题

对于外部合约优先使用 pull 而不是 push

在进行外部调用时，总会有意无意的失败，为了避免发生未知的损失，应该经可能的把对外的操作改为用户自己来取。 错误样例:

```
contract auction {
    address highestBidder;
    uint highestBid;
    function bid() payable {
```

```
            if (msg.value < highestBid) throw;
            if (highestBidder != 0) {
                if (!highestBidder.send(highestBid)) { // 可能会发生错误
                    throw;
                }
            }
            highestBidder = msg.sender;
            highestBid = msg.value;
        }
    }
```

当需要向某一方转账时，将转账改为定义 withdraw 函数，让用户自己来执行合约将余额取出，这样可以最大程度的避免未知的损失。

范例代码：

```
    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; // 记录在 refunds 中
```

```
            }
            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; //  如果转账错误还可以挽回
            }
        }
}
```

(6) 错误处理

合约中涉及到 call 等在 address 底层操作的方法时，做好合理的错误处理

address.call()

address.callcode()

address.delegatecall()

address.send()

这类操作如果遇到错误并不会抛出异常，而是会返回 false 并继续执行。

```
function withdraw(uint256 _amount) public {
    require(balances[msg.sender] >= _amount);
```

```
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
    }
```

上述代码没有校验 send 的返回值，如果 msg.sender 是合约账户，fallback 调用失败时，send 返回 false。

所以当使用上述方法时，需要对返回值做检查并做错误处理。

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

https://paper.seebug.org/607/#4-unchecked-return-values-for-low-level-calls

值得注意的一点是，作为 EVM 设计的一部分，下面这些函数如果调用的合约不存在，将会返回 True

call、delegatecall、callcode、staticcall

在调用这类函数之前，需要对地址的有效性做检查。

(7) 弱随机数问题

智能合约上随机数生成方式需要更多考量

Fomo3D 合约在空投奖励的随机数生成中就引入了 block 信息作为随机数种子生成的参数，导致随机数种子只受到合约地址影响，无法做到完全随机。

```
function airdrop()
    private
    view
```

```
        returns(bool)
    {
        uint256 seed = uint256(keccak256(abi.encodePacked(

            (block.timestamp).add
            (block.difficulty).add
            ((uint256(keccak256(abi.encodePacked(block.coinbase)))) / (now)).add
            (block.gaslimit).add
            ((uint256(keccak256(abi.encodePacked(msg.sender)))) / (now)).add
            (block.number)
        )));
        if((seed - ((seed / 1000) * 1000)) < airDropTracker_)
            return(true);
        else
            return(false);
    }
```

上述这段代码直接导致了 Fomo3d 薅羊毛事件的诞生。真实世界损失巨大，超过数千 eth。

所以在合约中关于这样的应用时，考虑更合适的生成方式和合理的利用顺序非常重要。

这里提供一个比较合理的随机数生成方式 hash-commit-reveal，即玩家提交行动计划，然后行动计划 hash 后提交给后端，后端生成相应的 hash 值，然后生成对应的随机

数 reveal，返回对应随机数 commit。这样，服务端拿不到行动计划，客户端也拿不到随机数。

有一个很棒的实现代码是 dice2win 的随机数生成代码。

但 hash-commit-reveal 最大的问题在于服务端会在用户提交之后短暂的获得整个过程中的所有数据，如果恶意进行选择中止攻击，也在一定程度上破坏了公平性。详细分析见智能合约游戏之殇——Dice2win 安全分析

当然 hash-commit 在一些简单场景下也是不错的实现方式。即玩家提交行动计划的 hash，然后生成随机数，然后提交行动计划。

真实世界事件：

Fomo3d 薅羊毛

https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide/

8 万笔交易「封死」以太坊网络，只为抢夺 Fomo3D 大奖？

Last Winner

https://paper.seebug.org/672/

(8) 变量覆盖问题

在合约中避免 array 变量 key 可以被控制

map[uint256(msg.sender)+x] = blockNum;

在 EVM 中数组和其他类型不同，因为数组时动态大小的，所以数组类型的数据计算方式为

address(map_data) = sha3(key)+offset

其中 key 就是 map 变量定义的位置，也就是 1，offset 就是数组中的偏移，比如 map[2]，offset 就是 2.

map[2]的地址就是 sha3(1)+2，假设 map[2]=2333，则 storage[sha3(1)+2]=2333。

这样一来就出现问题了，由于 offset 我们可控，我们就可以向 storage 的任意地址写值。

这就可能覆盖 storage 的任意地址的值，影响代码本身的逻辑，导致进一步更严重的问题。

详细的原理可以看

•以太坊智能合约 OPCODE 逆向之理论基础篇

•https://paper.seebug.org/739/

4.5 编码问题隐患

(1) 语法特性问题

在智能合约中小心整数除法的向下取整问题

在智能合约中，所有的整数除法都会向下取整到最接近的整数，当我们需要更高的精度时，我们需要使用乘数来加大这个数字。

该问题如果在代码中显式出现，编译器会提出问题警告，无法继续编译，但如果隐式出现，将会采取向下取整的处理方式。

错误样例

```
uint x = 5 / 2; // 2
```

正确代码

```
uint multiplier = 10;

uint x = (5 * multiplier) / 2;
```

(2) 数据私密问题

注意链上的所有数据都是公开的

在合约中，所有的数据包括私有变量都是公开的，不可以将任何有私密性的数据储存在链上。

(3) 数据可靠性

合约中不应该让时间戳参与到代码中，容易受到矿工的干扰，应使用 block.height 等不变的数据

```
uint someVariable = now + 1;

if (now % 2 == 0) { // now 可能被矿工控制

}
```

(4) gas 消耗优化

对于某些不涉及状态变化的函数和变量可以加 constant 来避免 gas 的消耗

```
contract EUXLinkToken is ERC20 {

    using SafeMath for uint256;

    address owner = msg.sender;

    mapping (address => uint256) balances;

    mapping (address => mapping (address => uint256)) allowed;

    mapping (address => bool) public blacklist;

    string public constant name = "xx";

    string public constant symbol = "xxx";

    uint public constant decimals = 8;

    uint256 public totalSupply = 1000000000e8;

    uint256 public totalDistributed = 200000000e8;
```

```
uint256 public totalPurchase = 200000000e8;

uint256 public totalRemaining = totalSupply.sub(totalDistributed).sub(totalPurchase);

uint256 public value = 5000e8;

uint256 public purchaseCardinal = 5000000e8;

uint256 public minPurchase = 0.001e18;

uint256 public maxPurchase = 10e18;
```

(5) 合约用户

合约中，应尽量考虑交易目标为合约时的情况，避免因此产生的各种恶意利用

```
contract Auction{

    address public currentLeader;

    uint256 public hidghestBid;

    function bid() public payable {

        require(msg.value > highestBid);

        require(currentLeader.send(highestBid));

        currentLeader = msg.sender;

        highestBid = msg.value;

    }

}
```

上述合约就是一个典型的没有考虑合约为用户时的情况，这是一个简单的竞拍争夺王位的代码。当交易 ether 大于合约内的 highestBid，当前用户就会成为合约当前的"王"，他的交易额也会成为新的 highestBid。

```
contract Attack {
    function () { revert(); }
    function Attack(address _target) payable {
        _target.call.value(msg.value)(bytes4(keccak256("bid()")));
    }
}
```

但当新的用户试图成为新的“王”时，当代码执行到require(currentLeader.send(highestBid));时，合约中的 fallback 函数会触发，如果攻击者在 fallback 函数中加入 revert()函数，那么交易就会返回 false，即永远无法完成交易，那么当前合约就会一直成为合约当前的"王"。

(6) 日志记录

关键事件应有 Event 记录，为了便于运维监控，除了转账，授权等函数以外，其他操作也需要加入详细的事件记录，如转移管理员权限、其他特殊的主功能

```
function transferOwnership(address newOwner) onlyOwner public {
    owner = newOwner;
    emit OwnershipTransferred(owner, newowner);
    }
```

(7) 回调函数

合约中定义 Fallback 函数，并使 Fallback 函数尽可能的简单

Fallback 会在合约执行发生问题时调用（如没有匹配的函数时），而且当调用 send 或者 transfer 函数时，只有 2300gas 用于失败后 fallback 函数执行,2300 gas 只允许执行一组字节码指令，需要谨慎编写，以免 gas 不够用。

部分样例：

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

function() public payable{ revert();};

(8) Owner 权限问题

避免 owner 权限过大

部分合约 owner 权限过大，owner 可以随意操作合约内各种数据，包括修改规则，任意转账，任意铸币烧币，一旦发生安全问题，可能会导致严重的结果。

关于 owner 权限问题，应该遵循几个要求： 1、合约创造后，任何人不能改变合约规则，包括规则参数大小等 2、只允许 owner 从合约中提取余额

(9) 用户鉴权问题

合约中不要使用 tx.origin 做鉴权

tx.origin 代表最初始的地址，如果用户 a 通过合约 b 调用了合约 c，对于合约 c 来说，tx.origin 就是用户 a，而 msg.sender 才是合约 b，对于鉴权来说，这是十分危险的，这代表着可能导致的钓鱼攻击。

下面是一个范例:

```
pragma solidity >0.4.24;

// THIS CONTRACT CONTAINS A BUG - DO NOT USE

contract TxUserWallet {

    address owner;

    constructor() public {

        owner = msg.sender;

    }

    function transferTo(address dest, uint amount) public {
```

```
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

我们可以构造攻击合约

```
pragma solidity >0.4.24;
interface TxUserWallet {
    function transferTo(address dest, uint amount) external;
}
contract TxAttackWallet {
    address owner;
    constructor() public {
        owner = msg.sender;
    }
    function() external {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

当用户被欺骗调用攻击合约，则会直接绕过鉴权而转账成功，这里应使用 msg.sender 来做权限判断。

https://solidity.readthedocs.io/en/develop/security-considerations.html#tx-origin

(10) 条件竞争问题

合约中尽量避免对交易顺序的依赖

在智能合约中，经常容易出现对交易顺序的依赖，如占山为王规则、或最后一个赢家规则。都是对交易顺序有比较强的依赖的设计规则，但以太坊本身的底层规则是基于矿工利益最大法则，在一定程度的极限情况下，只要攻击者付出足够的代价，他就可以一定程度控制交易的顺序。开发者应避免这个问题。

真实世界事件：

Fomo3d 事件

智能合约游戏之殇——类 Fomo3D 攻击分析

(11) 未初始化的储存指针

避免在函数中初始化 struct 变量

在 solidity 中允许一个特殊的数据结构为 struct 结构体，而函数内的局部变量默认使用 storage 或 memory 储存。

而存在 storage(存储器)和 memory(内存)是两个不同的概念，solidity 允许指针指向一个未初始化的引用，而未初始化的局部 stroage 会导致变量指向其他储存变量，导致变量覆盖，甚至其他更严重的后果。

```
pragma solidity ^0.4.0;

contract Test {

        address public owner;

        address public a;

        struct Seed {

                address x;

                uint256 y;
```

```
    }
    function Test() {
        owner = msg.sender;
        a = 0x1111111111111111111111111111111111111111;
    }
    function fake_foo(uint256 n) public {
        Seed s;
        s.x = msg.sender;
        s.y = n;
    }
}
```

上面代码编译后，s.x 和 s.y 会错误的指向 ownner 和 a。

攻击者在执行 fake_foo 之后，会将 owner 修改为自己。

上述问题在最新版的 0.4.25 版本被修复。

(12)「输入假名」问题

在引入零知识证明作为校验合约的条件时，需要谨防“输入假名”问题。

在密码学中，椭圆曲线的{x, y}的值域是一个基于 mod p 的有限域，基于椭圆曲线的加密也就在这个有限域的循环中进行。在智能合约的实现中，可能存在多个数经过 mod 运算后都会对应同一个 Fq，在这种情况下，我们把这多个大整数称为“输入假名”，而这些数互为假名。这也就意味着，互为假名的多个数可以重复的通过校验，利用 5 个“输入假名”可以将一笔钱重复花费 5 次。

真实世界事件

zkSNARK 合约「输入假名」漏洞致众多混币项目爆雷

5. 总结

智能合约发展十分迅猛。尽管一次次的安全事件不断刺激着人们的神经，但不可否认，智能合约正在往更丰富、更完善的方向发展。黑客的攻击从某种意义上来说也对这一进化起到了不可替代的增进作用。智能合约正变得越来越复杂多样，开发者可以借助区块链和智能合约完成更多有价值的应用，合约本身也将承载更高的价值。开发者和社区需要更多的安全力量投入来保障生态安全，而与智能合约和区块链安全相关的更多研究也正在路上。

参考资料

[1]. 以太坊智能合约 —— 最佳安全开发指南 [EB/OL].https://github.com/ConsenSys/smart-contract-best-practices/blob/master/README-zh.md,2019 年 10 月 21 日.

[2]. Contract Safety and Security Checklist[EB/OL].https://www.kingoftheether.com/contract-safety-checklist.html,.

[3]. 从 solidity 语言特性深度解读以太坊智能合约漏洞原理和攻击利用 [EB/OL].https://mp.weixin.qq.com/s/UXK8-ZN7mSUI3mPq2SC6Og,2018 年 8 月 2 日.

[4]. 以太坊智能合约重放攻击细节剖析 [EB/OL].https://mp.weixin.qq.com/s/kEGbx-I17kzm7bTgu-Nh2g,2018 年 8 月 16 日.

[5].https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20presentations/Bai%20Zheng%20and%20Chai%20Wang/DEFCON-26-Bai-Zheng-Chai-Wang-You-May-Have-Paid-more-than-You-Imagine.pdf

[6]. 以太坊智能合约安全入门了解一下（上） [EB/OL].http://rickgray.me/2018/05/17/ethereum-smart-contracts-vulnerabilites-review/,2018 年 5 月 17 日.

[7]. http://rickgray.me/2018/05/26/ethereum-smart-contracts-vulnerabilities-review-part2/

[8]. 区块链安全技术指南 黄连金，吴思进，曹锋，季宙栋 等，机械工业出版社，2018

邮箱:info@c-csa.cn
官网:https://c-csa.cn