

DApp

安全指南

@2020 云安全联盟大中华区-保留所有权利。你可以在你的电脑上下载、储存、展示、查看及打印，或者访问云安全联盟大中华区官网（<https://www.c-csa.cn>）。须遵守以下：(a) 本文只可作个人、信息获取、非商业用途；(b) 本文内容不得篡改；(c) 本文不得转发；(d) 该商标、版权或其他声明不得删除。在遵循中华人民共和国著作权法相关条款情况下合理使用本文内容，使用时请注明引用于云安全联盟大中华区。

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 DApp 安全小组专家撰写，并由 DApp 安全小组及区块链数据安全和数字钱包工作组的专家共同审核，感谢以下专家的贡献：

本白皮书贡献者名单：陈晶，黄连金，李岩，马红杰，王登辉，姚昌林，张志军，周庆松（按照字母排序）

贡献单位：知道创宇，武汉大学

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

去中心化应用（即 Decentralized Application，以下简称 DApp）作为区块链的重要实现载体。DApp 继承了传统应用的优势，结合区块链的特点，极大地扩展了区块链的应用场景与现实意义。DApp 可以被广泛地应用于金融(DeFi)、游戏、保险、物联网、共享经济、人工智能等多个领域。

但同时也面临着严重的安全风险，例如：2019 年 EOS DApp 安全成为重灾之地，截止 2019 年 5 月被盗 EOS 达到 93 万。相比于普通应用程序而言，DApp 的安全性不仅影响参与多方的公平性，还影响 DApp 所管理的庞大数字资产的安全性。因此对 DApp 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 DApp 的安全进行系统化研究，从不同的角度去分析 DApp 的安全。根据 2020 年初统计，DApps 应用中最受欢迎的是 DeFi，本文针对 DeFi 的安全应用场景进行了重点分析。



李雨航 Yale Li

CSA 大中华区主席兼研究院院长

目录

致谢.....	3
序言.....	4
1. DApp 与安全概述.....	6
1.1 DApp 概述.....	6
1.2 DApp 安全简介.....	7
1.3 DApp 安全攻防.....	8
2. DeFi 的安全.....	8
2.1 DeFi 概述.....	8
2.2 DeFi 的功能特性.....	9
2.3 DeFi 的生态堆栈.....	11
2.4 DeFi 项目的安全概述.....	12
2.5 DeFi 的智能合约安全案例分析.....	14
2.6 DeFi 通证经济设计方面的安全案例分析.....	26
2.7 DeFi 监管安全的案例分析.....	32
3. DApp 安全测试建议与工具.....	34
3.1 安全架构分析.....	34
3.2 静态代码扫描.....	35
3.3 动态应用扫描.....	36
3.4 手工渗透测试.....	37
3.5 生产环境中的测试与追踪.....	37
3.6 针对智能合约的安全测试工具.....	38
4. DApp 安全最佳实践与案例.....	39
4.1 DApp 的基本架构.....	39
4.2 安全开发与测试的要点.....	40
4.3 DApp 开发采用 DevSecOps 流程.....	41
4.4 加强安全审计.....	41
参考资料.....	43

1. DApp 与安全概述

1.1 DApp 概述

DApp 是 Decentralized Application 的缩写，是基于区块链的应用程序。这些类型的应用程序的身份数据和资产由用户控制，程序的运行和数据的管理不受任何中心化的管控。一个应用程序需要有一组特定的特征才能成为 DApp。具体包括：

去中心化：使用一种类似区块链的加密技术。不依赖中心服务器，不需要专门的通信服务器传递消息，也不需要中心数据库来管理数据。

开放生态系统：任何人都可以针对特定区块链平台的接口和协议开发应用程序，它的发布不受任何机构限制。各种创意与创新可以自由表达和实现。

算法及协议：交易发生的顺序由共识算法决定。DApp 参与者信息一般存储在参与者的手机。可以保护数字资产，有利于产权不会泄露、被破坏。

DApp 也在演变不断进化，通证化是今天 DApp 的显著特点。在区块链的时代，任何“价值”的流动，都可以通证化。通证需要具备三要素：权益、加密、流通，通证的三个要素缺一不可。通证化后的 DApp 只需聚焦在应用前端和智能合约之间的业务逻辑上。

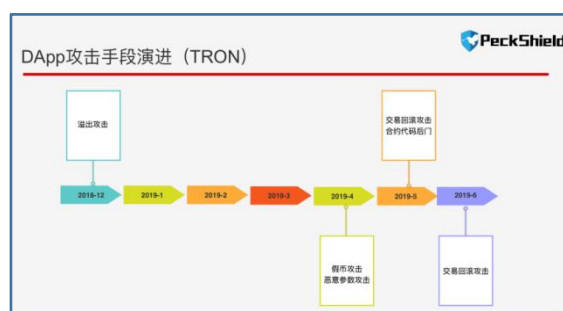
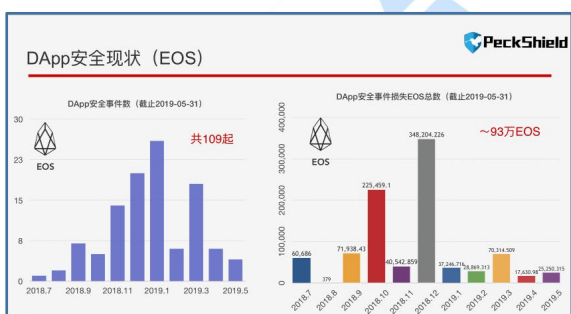
美国证监会（SEC）将加密货币划分为两类：**实用型通证（utility token）**和**证券型通证（security token）**。证券型通证被用来支付红利、收益、利息，或者通过投资其他通证为通证持有人带来利益。他们是具有货币价值的可交易金融工具。公共股本、私人股本、房地产、管理基金、交易所基金、债券，这些都是证券型通证的常见示例。这类通证也被细分为两类：权益通证（Equity Token）和资产通证（Asset Token）。实用型通证也被称为应用通证（app token）或用户通证（user token）。这类通证大多都是企业针对自己提供的服务或者产品为项目募资而发行的。实用型通证的价值是以项目概念的未来实用价值来评估的。与证券型通证相比，实用型通证更具体地强调自己的开发平台或生态系统，它的价值与平台或生态系统内参与者的活跃度成正比。实用型通证被细分为两类：产品或服务通证（Use of Product）和奖励通证（Reward Token）

1.2 DApp 安全简介

DApp 安全具有一些内置安全性、又可以通过密码学增加其鲁棒性。但由于 DApp 直接操作数字资产，而且往往开放源代码，应用程序的源代码是所有人都可见，已经成为备受黑客关注的攻击目标。黑客的攻击手段从最初的“溢出攻击”到“假 EOS 攻击”、“重放攻击”、“假转账通知攻击”，而后到屡试不爽的“随机数攻击”和“交易回滚攻击”，黑客的攻击手法正在不断演变且愈发复杂。以下是 Peckshield 发布的波场、以太坊、EOS 三大生态 DApp 安全事件报告：

根据 Peckshield 报告指出，从 2018 年 7 月至 2019 年 6 月，EOS 上的 DApp 安全事件数量最多，共 109 起，损失 EOS 总数约 93 万枚。波场 (Tron) 上的 DApp 安全事件数量为 11 起，损失 TRX 约 2900 万枚，而仅 2019 年 5 月波场发生的安全事件所引起是 TRX 损失就超 2700 万枚，占总统计数据的约 93%。

而以太坊安全事件（注：仅统计 DApp 的安全，不包括 Token（如常规 ERC20）相关的安全事件）有 5 起，共损失 2.4 万枚以太坊。下面的图介绍总结安全事件和攻击手段。



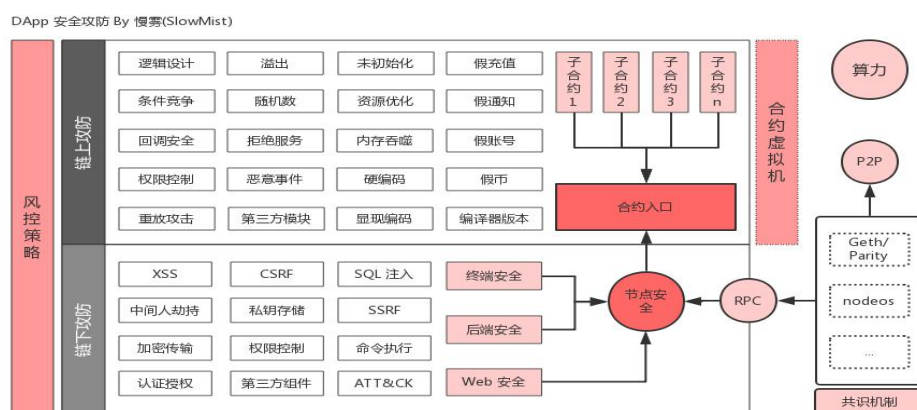
引自 (<https://www.theblockbeats.com/news/1959>)

1.3 DApp 安全攻防

根据区块链安全公司厦门慢雾科技有限公司（以下简称：慢雾科技）总结的安全攻防图对 DApp 的安全进行了总结。慢雾科技攻防图的总结，除了利用区块链上的常见的手段还总结了利用区块链下的传统安全的手段进行风险分析包括但不限于：

链上攻防：包括但不限于：逻辑设计、溢出、未初始化、假充值、条件竞争、随机数、资源优化、假通知、回调安全、拒绝服务、内存吞噬、假账号、权限控制、恶意事件、硬件编码、假币、重放攻击、第三方模块、显现编码、编译器版本等问题。

链下攻防：包括但不限于 XSS、CSRF、SQL 注入、中间人劫持、私钥存储、SSRF、加密传输、权限控制、命令执行、认证授权、第三方组件、ATT&CK、终端安全、后端安全、Web 安全、节点安全等问题，除了考虑智能合约安全，还需要考虑节点的安全。除了考虑数据的加密还需要考虑共识算法的安全。



(https://github.com/slowmist/Knowledge-Base/blob/master/dapp_attack_defense.png)

2. DeFi 的安全

2.1 DeFi 概述

DeFi 是 Decentralized Finance 的缩写（以下简称），可以翻译为去中心化金融，也

可以理解为开放式金融（Open Finance），它是利用去中心化协议和智能合约提供普惠的、不需要准入条件的金融服务。

由于区块链的金融属性，全球参与属性和 DAO（社区自治）属性，DApp 的最重要的应用就是 DeFi.应该说，2020 年区块链整个行业的发展是聚焦在 DeFi 的应用。

DeFi 是相对于 CeFi（传统的中心化金融，比如银行）而言的，我们现有的银行业、保险业、信托业、证券业、租赁业等几乎都是 CeFi，它们能不能满足我们的需求？可以，但是也存在下面例举的痛点。

1. 不够普惠性：以门槛最低的银行账户为例，据不完全统计，目前全球大约仍有 17 亿人没有银行账户。其中东南亚地区大约 4.6 亿人、非洲大约有 3.5 亿人、中国大约有 2.25 亿人都是没有银行账号的。即便在美国这个最发达的资本主义国家，目前仍有大约 5500 万人没有银行账号。

2. 费用高：无论是 CeFi 还是 DeFi，我们在使用各种金融服务时，一般都会产生一定的费用。但在 CeFi 中，这个费用有时会很高昂。以跨境转账为例，目前每笔的平均服务费用是 4.4%+0.3 美元。也就是说如果从中国汇款 100 万元美元到美国，我们需要支付高达 44000.3 美元的手续费。

再以贷款为例，大额贷款一般都要收取发放费、手续费，在美国这个费率一般是 2%。比如，贷款 100 万元，一般就要支付 2 万元左右的费用。

3. 效率低：还是以跨境转账为例，目前国际上普遍使用 SWIFT 平台，其转账的周期是 T+2，也就是在一笔交易完成后，还需要再等 2 天的时间才能到账。

4. 中心化安全问题：这个问题不言而喻，传统金融机构都会有中心化的数据库，一旦出现黑客拖库后果将不堪设想。当然现在银行等金融机构的安全防控级别都很高，但中心化的数据存储在机构作恶成为可能，比如用户隐私泄露早已成为社会的顽疾，恶意篡改用户数据也不无可能。

2.2 DeFi 的功能特性

针对 CeFi 的不足进行了理解，那么 DeFi 相较之下有哪些特性？以下是 DeFi 拥有的

10 个特性：

1. **普惠性：**只要一部可以联网的手机或者电脑，在世界任何一个角落任何一个人，都可以访问 DeFi 协议，从而享受各种的金融服务。比如用 Uniswap 进行去中心化交易，比如利用 Compound 平台进行去中心化借贷，或者通过 ImToken 钱包访问 DeFi 应用进行去中心化理财等。

2. **可控性：**相较于 CeFi 将资产交给金融机构托管，DeFi 的一大优势是用户始终对自己的资产拥有绝对的控制权。并且在 DeFi 中，用户在金融交易中可以任意扮演买方或卖方，既可以是服务方也可以是被服务方，而且没有什么门槛。

3. **透明性：**DeFi 协议代码都是开源的，因此任何人都可以验证其安全性、交互规则以及实际网络使用情况。

4. **抗审查性：**在 DeFi 中系统对所有的交易都是一视同仁的，同时还能保护交易者的隐私。

5. **可编程性：**DeFi 是通过智能合约来实现的，所以可以根据功能需要对智能合约进行编程调整，从而实现不同的金融逻辑。

6. **可组合性：**可以利用不同的底层协议与上层应用进行组合，从而产生各种新的应用。

7. **可互操作性：**DeFi 可以将数字资产与法币、物理资产的价值进行交互。比如在以太坊上的 Synthetix 项目就可以实现这一点，这种可以交互的资产也称之为合成资产。

8. **高利息性：**在传统中心化金融中，我们将把钱存入银行，或者放贷得到的利息往往非常低，甚至在某些国家还可能是负利息。相较而言，在 DeFi 生态中的利息就要高的多。一个好的项目会将贷款发放费、交易费拿出来作为利息，分发给参与者。

9. **零费用性：**在传统金融中，除去交易费用外，往往还存在账户费用。比如信用卡的年费，储蓄卡存款额度很小时，银行会收取小额管理费等等。

另外，前段时间美国石油期货价格出现了史无前例的负价格，其根本原因也是因为存储、运输都需要费用，从而导致期货价格为负的可能。但在 DeFi 中即便你的账户里

只有 0.000001 个 ETH，也不需要支付任何账户费用。而且根据数字货币的属性，它的价格也不可能出现负值。

10. 可分散性：DeFi 可以分散风险。例如在今年新冠疫情等因素的影响下，全球资产市场出现危机。我们可以将传统投资与数字货币进行组合，从而分散投资风险。

综上所述，相较于 CeFi 而言，DeFi 拥有众多独特的属性与优势，并且随着技术的不断成熟，其应用范围将会越来越广泛，甚至带来整个金融行业的革命。也正是基于这些原因，即便目前 DeFi 并不够成熟，但其发展与前景依然被广泛看好。

2.3 DeFi 的生态堆栈

从技术、功能角度，DeFi 项目可以划分为不同的层次，比如公链、钱包属于底层技术，DEX、借贷属于不同层次的应用。同时，不同层次可以相互组合产生新的应用。目前行业内很少有人去做这个划分，我们初步把 DeFi 的生态应用划分为 6 个层次：

1. 底层技术：包括公链（比如以太坊、元界 DNA）和钱包（比如库神，imToken，元界钱包）等等。底层公链技术主要为 DeFi 生态提供价值存储和价值共识的基础。

2. DEX（去中心化交易所）：比如 Uniswap、OX、Kyber 等等。去中心化交易所必须基于某一种公链上面，提供价值交换的平台。所有价值交换的记录都需要存储在一条或者多条公链上。

3. 稳定币：比如 USDT、USDC，PAX 等数字货币。稳定币给去中心化交易所提供可以锚定法定货币的交易对。为 DeFi 整个生态提供了与现实世界对应的计量单位。稳定币的发行一般都是通过在某一条公链上部署智能合约进行实现，并且在交易和支付方面发挥使用价值，因此稳定币是基于公链和交易基础上的 DeFi 应用。

4. 借贷和闪贷（Flash Loan）：比如 MarkerDAO、Compound、BzX、AAVE 等 DeFi 项目都包括借贷或者闪贷服务。DeFi 的借币方主要是利用某一种数字货币作为抵押，借出另外一种数字货币（大部分的应用场景是稳定币的借贷）进行交易获得利润。因此 DeFi 的借贷服务目前是建立在公链技术，去中心化交易所和稳定币基础上的金融服务。

5. ETF、合成资产、ABS 等金融衍生品：不仅是数字资产的衍生产品，也可以将物

理资产用数字资产的形式来表现。DeFi 的金融衍生品目前还是比较初期的应用，项目包括 synthetix.io 等等。这个层次的应用利用了前面 4 个层次的服务。

6. 预测市场、保险、供应链金融：这个层次的应用利用前面五个层次的服务，给 DeFi 在现实世界的金融服务落地提供更加广阔的想象空间。例如 Nexus Mutual，Cover 协议，Opyn 等等分布式保险协议利用公链技术实现保险服务，利用 DEX 交易保险的治理币和保险标的的数字货币，利用稳定币给保险产品进行定价，利用 Wrapped Token 进行借贷服务，利用合成资产与保险产品进行合成生成新的保险产品。

目前 DeFi 还不够成熟，暂时缺少成熟的具体的应用，但是发展前景广阔。需要注意的是，从第二到第六个层次，同时需要预言机（Oracle）的协助。并且不同层次的组合必须要注意安全问题。近期一个典型的案例就是 Lendf.Me 黑客攻击事件，其原因就是 ERC-777 与 Uniswap 组合所产生的问题。单独拿 ERC-777 与 Uniswap 来说，它们本身都没有什么问题，但组合在一起就可能产生一系列连锁反应，从而带来难以估测的安全隐患。所以在实际应用中需要进行大量测试，以确保应用的安全。

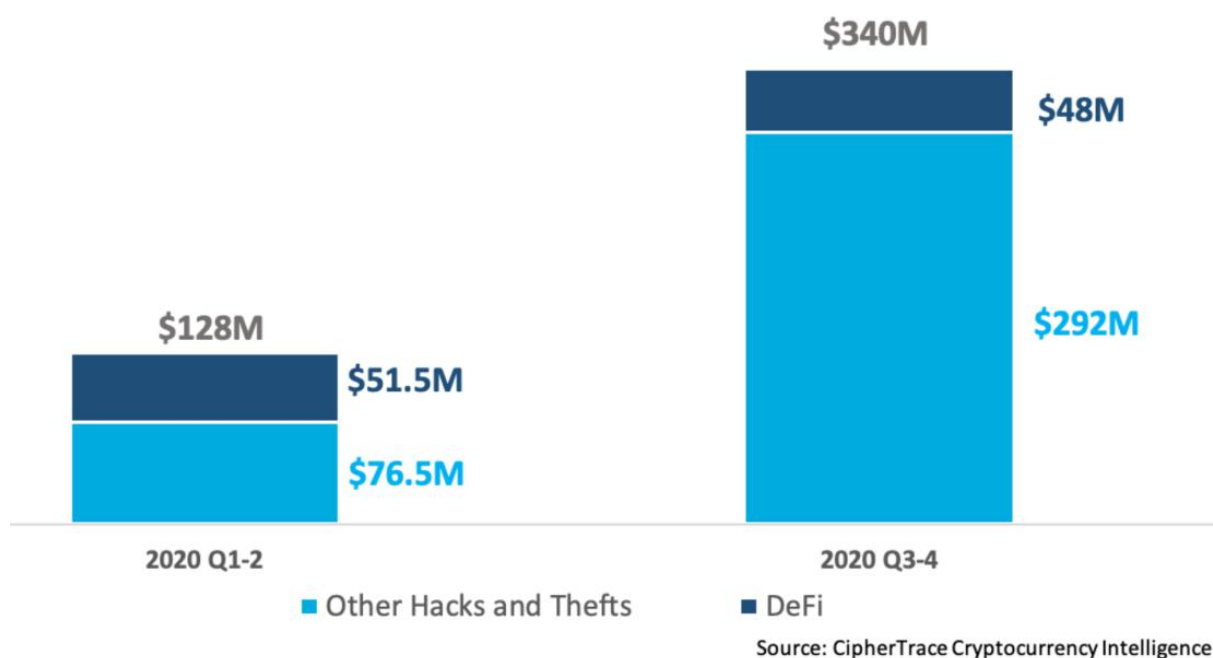
DeFi 目前最大的问题就是它还处于发展初期，各方面还不够成熟。像 MakerDAO、Uniswap 等知名项目其实大部分都还处于 Beta 阶段，出现包括漏洞在内的安全问题在所难免。

关于 DeFi 生态，目前大部分应用都是建立在以太坊公链之上，在其他公链，比如 BTC、EOS、波场、元界 DNA 上面也有，但相较于以太坊还是要少。应用类型方面，目前以借贷、合成资产与 DEX 为主。

2.4 DeFi 项目的安全概述

根据美国区块链数据分析公司 CipherTrace 统计，2020 年 DeFi 项目被黑客攻击造成的损失是 1 亿美金。

DeFi Adds \$100 Million to 2020 Thefts



我们总结，DeFi 项目需要注意三方面的安全，第一就是智能合约的安全，第二就是通证经济设计方面的安全，第三就是监管的安全。

DeFi 第一方面的安全：关于智能合约的安全，机械工业出版社在 2018 年 5 月出版的《区块链安全技术指南》本书里面，很大篇幅的内容讲智能合约的安全值得借鉴，另外云安全联盟 CSA 大中华区的区块链安全工作组也在最近发布了智能合约安全方面的白皮书也可以参考。

举两个例子来说明一下。第一个，是 DeFi 去中心化保险项目 Oryn 的漏洞，第二是波场与以太坊跨链的 DeFi 项目的漏洞。两个漏洞都与智能合约代码有关。

DeFi 第二方面的安全：就是通证经济设计方面的安全。如果通证经济没有设计好，会造成死亡螺旋的问题。就是你价格越低，参与的人越少，然后逐渐的就价格就归零了，或者趋近于零，这就是死亡螺旋，你怎么样去避免死亡螺旋，促进价格和生态可持续发展，这个其实是通证经济与 DAO 设计的一个关键。或者因为通证经济设计参数方面有漏洞，可以被黑客利用。我们下面会用 DeFi 项目 Curve 作为例子来说明可能被黑客利用的通证经济设计参数方面的漏洞。总体来说 DeFi 的 90% 的项目，因为有可能会因为共识不够，通证经济设计不合理，可能技术还可以，但是最后还是要靠生态，靠通证

经济，因为它是多学科的领域，其中某个领域没做好，最后可能不能成功偃旗息鼓了，这些例子已经非常多，就不一一列举了。

DeFi 第三方面的安全：是监管安全。现在 DeFi 生态还小，监管还没有开始。但是到某个程度就要受监管，那么有些如果不符合监管的话，整个生态会受到打击，所以这个风险就严重了。下面我们用以中心化交易所 EtherDelta 项目被美国政府罚款作为例子进行分析说明。需要注意的是 DeFi 项目最终都一定要符合本地的监管，所以首先项目需要有自律的精神，绝对不能够去做非法的一些事情。DeFi 要能够真正地进行发展，一定要有行业的自律，现在国内很多 DeFi 的仿盘，内在还是中心化的，不是去中心化，有可能会圈钱跑路的，所以大家要小心，保证好项目的安全工作，及时规避风险。

2.5 DeFi 的智能合约安全案例分析

DeFi 的业务逻辑由智能合约的代码实现，智能合约本身的安全可以参考 CSA GCR 的智能合约安全指南白皮书，也可以参考机械工业出版社的区块链安全技术指南这本书关于智能合约安全的内容。这里主要具体分析两个智能合约安全被黑客攻击成功的案例。另外下面是与智能合约安全有关的需要考虑的内容。

1. 私钥风险：没有多签的 DeFi 合约意味着掌握合约私钥的可以随意更改合约或者跑路；因此 DeFi 合约的管理地址一定需要多签名。但是目前大部分 DeFi 项目的管理地址是单签名，或者不能证明多签名地址是一个人还是多个人拥有。随着 DeFi 应用增加这个问题会越来越重要。

2. 无常损失风险：例如自动做市的流动性挖矿提供者的无常损失，尤其是如果交易对的两种风险资产的价格波动太大，流动性提供者按照智能合约的自动做市的逻辑就会有大的无常损失。这个需要优化智能合约的自动做市的逻辑，预言机的价格公正性和可用性，和交易对价格的稳定性来解决。

3. 交易摩擦风险：在以太坊交易 Gas 费率极高的时候，DeFi 用户必须支付非常高的费用，如果本金小，就可能在不知不觉中损失本金。这个需要设计 Gas 费用合理的新的公链，或者第二层协议来解决 Gas 费用高的问题。

4. 操作失误风险：在转账过程中失误导致资产永久丢失。比如直接向 DeFi 智能合

约地址转账，造成资金被永远锁定在合约中。这个需要用户对于智能合约和 DeFi 项目本身的了解。

5. 闪贷（Flash Loan）造成的 DeFi 风险：闪贷的基本工作原理是：在单笔交易中贷出借款人需要的金额。然而在交易结束时，借款人必须偿还不少于贷款金额的数目。如果借款人做不到，贷款智能合约的逻辑会自动回滚交易。黑客利用闪贷零成本借贷，并且利用所借到的资本通过交易手段拉高或者拉低某一种数字货币进行价格操纵，把借来的币用高价格卖出，再用低价格买进以后全额还回。这些操作是通过一系列的智能合约调用在同一个区块完成，黑客单笔交易中获利。闪贷造成的 DeFi 风险本身不是因为有闪贷才造成的，而是由于 DeFi 项目本身在流动性设计或者预言机选择的漏洞造成的。理论上黑客通过闪贷调用智能合约的获利，都可以被一个拥有充分多数字货币的大户通过同样的智能合约调用来获取。因此解决闪贷（Flash Loan）造成的 DeFi 风险还是需要看 DeFi 项目本身的流动性设计和预言机设计。

6. 智能合约授权滥用风险：DeFi 项目比如 Compound, Uniswap, Kyber, Maker 等都是通过 ERC20 的 API Approve 获得用户的一次性授权，但是授权的金额没有限制，理论上这些项目可以滥用权限转走用户授权的资金。这个问题目前没有收到大家的重视。解决这个问题的方法非常简单，就是项目只能获得用户有明确限额的授权。每次超过限额的调用都需要用户批准。

2.5.1 Opyn 的智能合约漏洞分析

北京时间 2020 年 08 月 05 日，DeFi 期权平台和保险项目 Opyn 的看跌期权（Opyn ETH Put）智能合约遭到黑客攻击，损失约 37 万美元。

Opyn 是一个通用期权协议，于今年 2 月份转型为保险平台，通过 oTokens 为 DeFi 平台提供可交易的 ETH 看跌期权，以此锚定 ETH 市场价格，为高波动性的 DeFi 市场提供相对的稳定性。PeckShield 安全团队获悉 Opyn 平台遭受攻击后，迅速定位到问题关键点在于：

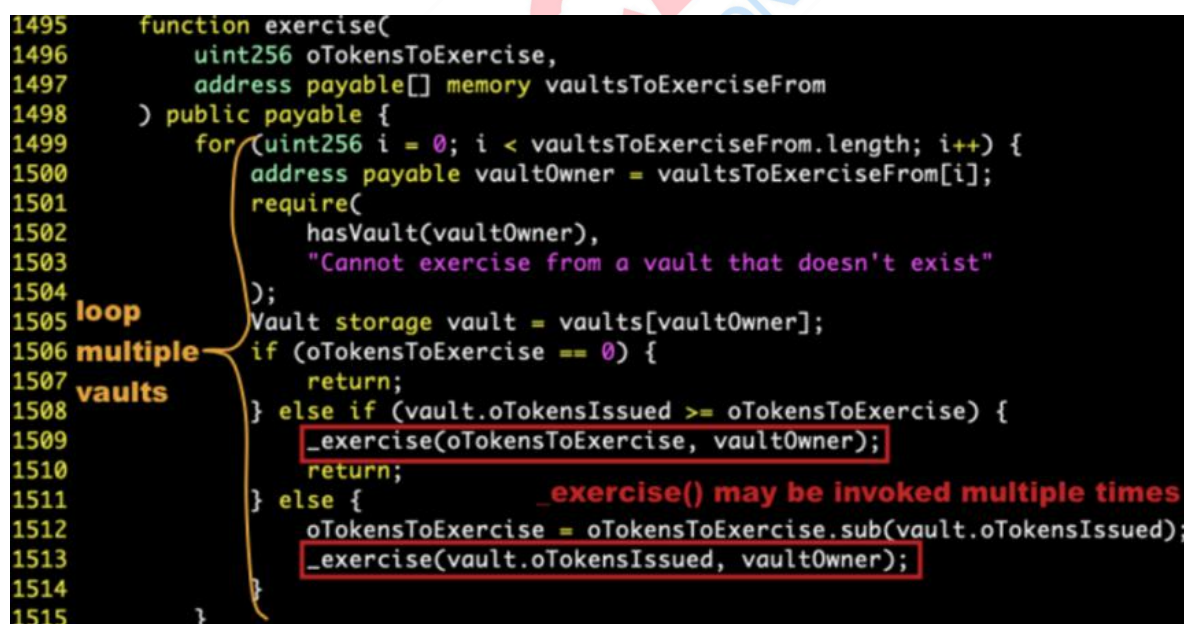
攻击者发现 Opyn 智能合约行权（exercise）接口对接收到的 ETH 存在某些处理缺陷，其合约并没有对交易者的实时交易额进行检验，使得攻击者可以在一笔对自己发起真实的交易之后，再插入一笔伪装交易骗得卖方所抵押的数字资产，进而实现空手套

白狼。

简单来说，由于 Opyn ETH Put 智能合约中的行权函数 `exercise()` 没有对交易者的 ETH 进行实时校验。根据 Opyn 平台的业务逻辑，看跌期权的买方给卖方转移相应价值的 ETH，即可获得卖方抵押的数字资产。狡猾的攻击者，先向自己发起伪装的交易，利用这笔 ETH 可以重复使用的特性，再次向卖方用户发起转账，进而骗取卖方已经抵押的数字资产。

下面是 PeckShield 的详细分析：

漏洞详细过程分析：先来说说，Opyn 平台的业务逻辑：当用户使用 Opyn 合约行权即买卖期货（`exercise`）时，需要买方向卖方转入相应数量的 ETH 或者 ERC20 Token，然后合约将销毁买方对应的 `oToken`，而后买方将获得卖方已经抵押的资产。例如：小王认为行情进入了下跌趋势，看到 Opyn 上挂着一个小李对 ETH 330 美元的看跌期权，于是进入交易系统，向小李转账一个 ETH，获得小李抵押的等额数字资产。若此刻行情已经跌至了 300 美元，小王便可获得其中的差价。



```
1495 function exercise(  
1496     uint256 oTokensToExercise,  
1497     address payable[] memory vaultsToExerciseFrom  
1498 ) public payable {  
1499     for (uint256 i = 0; i < vaultsToExerciseFrom.length; i++) {  
1500         address payable vaultOwner = vaultsToExerciseFrom[i];  
1501         require(  
1502             hasVault(vaultOwner),  
1503             "Cannot exercise from a vault that doesn't exist"  
1504         );  
1505         Vault storage vault = vaults[vaultOwner];  
1506         if (oTokensToExercise == 0) {  
1507             return;  
1508         } else if (vault.oTokensIssued >= oTokensToExercise) {  
1509             _exercise(oTokensToExercise, vaultOwner);  
1510             return;  
1511         } else {  
1512             oTokensToExercise = oTokensToExercise.sub(vault.oTokensIssued);  
1513             _exercise(vault.oTokensIssued, vaultOwner);  
1514         }  
1515     }  
}
```

图 1 `exercise()` 函数中循环执行传入的 `vaults` 地址列表

如上面的合约代码片段所示，行权函数 `exercise()` 的内部是一个循环，依据参数中传递的 `vaultsToExerciseFrom` 中的地址数量依次调用真正的行权逻辑 `_exercise()` 函数。

```

1876 // 4. Transfer in underlying, burn oTokens + pay out collateral
1877 // 4.1 Transfer in underlying
1878 if (isETH(underlying)) {
1879     require(msg.value == amtUnderlyingToPay, "Incorrect msg.value");
1880 } else {
1881     require(
1882         underlying.transferFrom(
1883             msg.sender,
1884             address(this),
1885             amtUnderlyingToPay
1886         ),
1887         "Could not transfer in tokens"
1888     );
1889 }

```

ERC20 case takes in tokens in each call

msg.value of ETH is re-used in the second (and further) _exercise call

图 2 重用传入合约的 ETH 来获得抵押资产

函数处理 ERC20 Token 时，和大部分的 DeFi 项目做法一样，使用 `transferFrom()`，如代码 1882 行所示，从 `msg.sender` 转账到 `address(this)`。

但是当函数处理的资产为 ETH 时，处理的方式就完全不一样了。因为在 Solidity 中，`msg.value` 的意思是合约调用者在调用具有 `payable` 接口时所转给该合约的 ETH 数量，仅是一个量值，所以在合约代码的 1879 行中，检查 `msg.value == amtUnderlyingToPay` 仅能确保合约确实收到了 `amtUnderlyingToPay` 数量的 ETH，并不会对 `msg.value` 的值造成任何影响。

但是正如上面讲到的在 `exercise()` 中会循环调用 `_exercise()` 函数，这导致尽管合约实际只收到一次 ETH，然而在循环过程中却可以重复使用。

攻击点就在这里，由于合约少了一步对 ETH 实时数量的检验，使得攻击者可以先伪造一笔指向自己的交易，然后再把已经花掉的本金再次利用，和平台其他用户完成一笔正常交易

```

call #fad517ac for Smart Contract 0xe7870231992ab4b...
  → addERC20CollateralOption ( amtToCreate: 750000000 , amtCollateral: 24750000000 , receiver: e7870231992ab4b1a... ) for Token oETH $330 Put 08/14/20
  → transferFrom ( from: e7870231992ab4b1a... , to: 951d51baefb72319d... , value: 24750000000 ) for Smart Contract FiatTokenProxy, Stablecoin,
https://www.centre.io/, USD Coin (USDC), USD Coin, Token Contract
  → transferFrom ( from: e7870231992ab4b1a... , to: 951d51baefb72319d... , value: 24750000000 ) delegate call to Token FiatTokenV1
  → exercise ( oTokensToExercise: 1500000000 , vaultsToExerciseFrom: [e7870231992ab4b1a..., 25125e438b7ae0f9a..., ] ) > transfer 75.0ETH to Token oETH $330
Put 08/14/20
  → transfer ( to: e7870231992ab4b1a... , value: 24750000000 ) for Smart Contract FiatTokenProxy, Stablecoin, https://www.centre.io/, USD Coin (USDC), USD Coin,
Token Contract
    first 24,750 USDC out, 75 oETH burned
  → transfer ( to: e7870231992ab4b1a... , value: 24750000000 ) delegate call to Token FiatTokenV1
  → transfer ( to: e7870231992ab4b1a... , value: 24750000000 ) for Smart Contract FiatTokenProxy, Stablecoin, https://www.centre.io/, USD Coin (USDC), USD Coin,
Token Contract
    second 24,750 USDC out, 75 oETH burned
  → transfer ( to: e7870231992ab4b1a... , value: 24750000000 ) delegate call to Token FiatTokenV1
  → removeUnderlying ( ) for Token oETH $330 Put 08/14/20
  → call > transfer 75.0ETH to Smart Contract 0xe7870231992ab4b...

```

two vaults to exercise()

msg.value is 75 ETH

图 3 攻击交易分析

在图 3 中，我们通过 Bloxy 浏览器显示的调用过程来展示攻击的过程。由于攻击者吃掉了很多笔订单，我们以其中一笔交易为例，向大家展示其攻击逻辑：

1. 攻击者先从 Uniswap 购入了 75 oETH 为进一步调用函数行权做好筹备；
2. 攻击者创建了一个 Vault 地址，作为看空期权卖方，并且抵押 24,750 USDC 铸造出 75 oETH，但并未卖出这些期权，等于自己同时买入了以 330 的价格卖出 75 ETH 的权利；
3. 攻击者在 Oryn 合约中调用了 exercise()，在持有 150 oETH 看空期权的情况下，先向自己的 Vault 地址转入了 75 个 ETH，获得自己事先抵押的 24,750 个 USDC，再重利用了这 75 个 ETH，成功吃掉了另一个用户的 24,750 个 USDC，进而实现非法获利。

修复建议

PeckShield 安全团队建议，在 Solidity 中，合约可使用一个局部变量 msgValue 来保存所收到 ETH（即 msg.value 的值）。这样，在后续的步骤中通过操作 msgValue，就能准确的标记有多少 ETH 已经被花费，进而避免资产被重复利用。此外，我们还可以使用 address(this).balance 来检查合约余额来规避 msg.value 被重复使用的风险。

2.5.2 波场 DeFi 项目 Myrose 智能合约漏洞分析

1. 事件起因

2020 年 9 月 14 日晚 20:00 点，未经安全审计的波场最新 DeFi 项目 Myrose.finance 登陆 Tokenpocket 钱包，首批支持 JST、USDT、SUN、DACC 挖矿，并将逐步开通 ZEUS、PEARL、CRT 等的挖矿，整个挖矿周期将共计产出 8400 枚 ROSE，预计将分发给至少 3000 名矿工，ROSE 定位于波场 DeFi 领域的基础资产。项目上线之后引来了众多的用户(高达 5700 多人)参与挖矿，但是好景不长，在 20:09 左右有用户在 Telegram"Rose 中文社区群"中发文表示 USDT 无法提现。

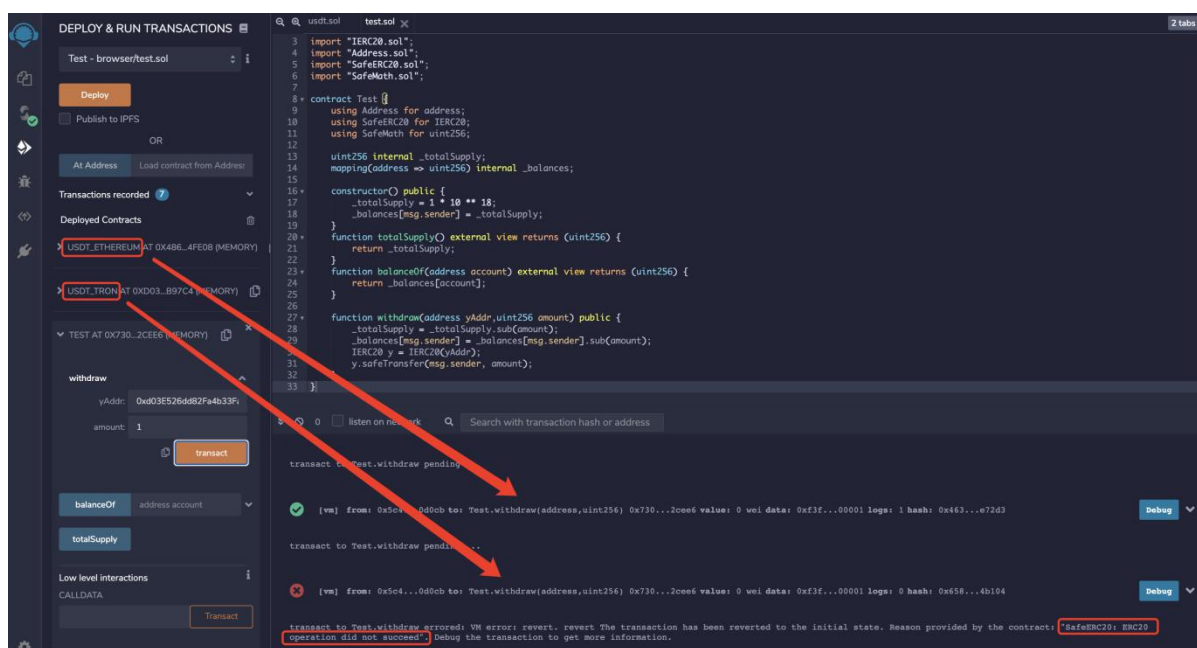
CSA 大中华区的区块链安全工作组成员知道创宇 404 区块链安全研究团队对于整个事件进行分析，描述如下：

2. 代码的分析:

知道创宇在 remix 上部署和测试 USDT_Ethereum、USDT_Trone、Test 三个合约（合约代码具体链接：<https://paper.seebug.org/1337/>）

调用 USDT_Ethereum 和 USDT_Trone 的 mint 函数给 Test 合约地址增添一些代币。

然后调用 Test 合约的 withdraw 函数提现测试。



可以看到 USDT_Ethereum 提现成功，USDT_Trone 提现失败。

失败的回滚信息中，正是 safeTransfer 函数中对最后返回值的校验。

知道创宇的合约模拟实验揭示了以太坊与波场两个不同平台下 USDT 代币合约中 transfer 函数关于返回值处理差异性带来的安全风险，而关于"missing return value bug"这一个问题，早在 2018 年就有研究人员在 Medium 上公开讨论过，只不过是针对以太坊的，这里对以太坊中的"missing return value bug"问题做一个简单的介绍：

3. Missing Return Value Bug

ERC20 标准是以太坊平台上最常见的 Token 标准，ERC20 被定义为一个接口，该接口指定在符合 ERC20 的智能合同中必须实现哪些功能和事件。

在 ERC20 的开发过程中，有研究人员对于 ERC20 合约中的 transfer 函数的正确返回

值进行了讨论，主要分为两个阵营：一方认为，如果 `transfer` 函数允许在调用合约中处理 `Failed error`，那么应该在被调用合约中返回 `false` 值，另一方声称，在无法确保安全的情况下，ERC20 应该 `revert` 交易，关于这个问题在当时被认为都是符合 ERC20 标准的，并未达成一致。

事实证明，很大比例的 ERC20 Token 在传递函数的返回值方面表现出了另一种特殊的方式，有些智能合约的 `Transfer` 函数不返回任何东西。

那么符合 ERC20 标准的接口的合约试图与不符合 ERC20 的合约进行交互，会发生什么呢？下面我们通过一个合约示例来做解释说明：

```
interface Token {  
  
    function transfer() returns (bool);  
  
}  
  
contract GoodToken is Token {  
  
    function transfer() returns (bool) { return true; }  
  
}  
  
contract BadToken {  
  
    function transfer() {}  
  
}  
  
contract Wallet {  
  
    function transfer(address token) {  
  
        require(Token(token).transfer());  
  
    }  
  
}
```

在 solidity 中，函数选择器是从它的函数名和输入参数的类型中派生出来的：

```
selector = bytes4(sha3("transfer()"))
```

函数的返回值不是函数选择器的一部分，因此，没有返回值的函数 `transfer()` 和函数 `transfer()` 返回(`bool`)具有相同的函数选择器，但它们仍然不同，由于缺少返回值，编译器不会接受 `transfer()` 函数作为令牌接口的实现，所以 `Goodtoken` 是 `Token` 接口的实现，而 `Badtoken` 不是。

当我们通过合约去外部调用 `BadToken` 时，`Bad token` 会处理该 `transfer` 调用，并且不返回布尔返回值，之后调用合约会在内存中查找返回值，但是由于被调用的合约中的 `Transfer` 函数没有写返回值，所以它会将在这个内存位置找到的任何内容作为外部调用的返回值。

完全巧合的是，因为调用方期望返回值的内存槽与存储调用的函数选择器的内存槽重叠，这被 `EVM` 解释为返回值“真”。因此，完全是运气使然，`EVM` 的表现就像程序员们希望它的表现一样。

自从 2018 年 10 月以太坊拜占庭硬分叉以来，`EVM` 有了一个新的操作码，叫做 `return data size`，这个操作码存储(顾名思义)外部调用返回数据的大小，这是一个非常有用的操作码，因为它允许在函数调用中返回动态大小的数组。

这个操作码在 `solidity 0.4.22` 更新中被采用，现在，代码在外部调用后检查返回值的大小，并在返回数据比预期的短的情况下 `revert` 事务，这比从某个内存插槽中读取数据安全得多，但是这种新的行为对于我们的 `BadToken` 来说是一个巨大的问题。

如上所述，最大的风险是用 `solc ≥ 0.4.22` 编译的智能合约(预期为 `ERC0` 接口)将无法与我们的 `Badtokens` 交互，这可能意味着发送到这样的合约的 `Token` 将永远停留在那里，即使该合约具有转移 `ERC 20 Token` 的功能。

类似问题的合约：

```
{'addr': '0xae616e72d3d89e847f74e8ace41ca68bbf56af79', 'name': 'GOOD', 'decimals':  
6}
```

```
{'addr': '0x93e682107d1e9defb0b5ee701c71707a4b2e46bc', 'name': 'MCAP', 'decimals':  
8}  
  
{'addr': '0xb97048628db6b661d4c2aa833e95dbe1a905b280', 'name': 'PAY', 'decimals':  
18}  
  
{'addr': '0x4470bb87d77b963a013db939be332f927f2b992e', 'name': 'ADX', 'decimals':  
4}  
  
{'addr': '0xd26114cd6ee289accf82350c8d8487fedb8a0c07', 'name': 'OMG', 'decimals':  
18}  
  
{'addr': '0xb8c77482e45f1f44de1745f52c74426c631bdd52', 'name': 'BNB', 'decimals':  
18}  
  
{'addr': '0xf433089366899d83a9f26a773d59ec7ecf30355e', 'name': 'MTL', 'decimals':  
8}  
  
{'addr': '0xe3818504c1b32bf1557b16c238b2e01fd3149c17', 'name': 'PLR', 'decimals':  
18}  
  
{'addr': '0xe2e6d4be086c6938b53b22144855eef674281639', 'name': 'LNK', 'decimals':  
18}  
  
{'addr': '0x2bdc0d42996017fce214b21607a515da41a9e0c5', 'name': 'SKIN', 'decimals':  
6}  
  
{'addr': '0xea1f346faf023f974eb5adaf088bbcdf02d761f4', 'name': 'TIX', 'decimals': 18}  
  
{'addr': '0x177d39ac676ed1c67a2b268ad7f1e58826e5b0af', 'name': 'CDT', 'decimals':  
18}
```

有两种方法可以修复这个错误:

第一种: 受影响的 Token 合约开放团队需要修改他们的合约, 这可以通过重新部署 Token 合约或者更新合约来完成(如果有合约更新逻辑设计)。

第二种：重新包装 Bad Transfer 函数，对于这种包装有不同的建议，例如：

```
library ERC20SafeTransfer {

    function safeTransfer(address _tokenAddress, address _to, uint256 _value) internal
    returns (bool success) {

        // note: both of these could be replaced with manual mstore's to reduce cost
        if desired

        bytes memory msg = abi.encodeWithSignature("transfer(address,uint256)",
        _to, _value);

        uint msgSize = msg.length;

        assembly {

            // pre-set scratch space to all bits set

            mstore(0x00, 0xff)

            // note: this requires tangerine whistle compatible EVM

            if iszero(call(gas(), _tokenAddress, 0, add(msg, 0x20), msgSize, 0x00,
0x20)) { revert(0, 0) }

            switch mload(0x00)

            case 0xff {

                // token is not fully ERC20 compatible, didn't return anything,
                assume it was successful

                success := 1
            }
        }
    }
}
```



```

    }

    case 0x01 {

        success := 1

    }

    case 0x00 {

        success := 0

    }

    default {

        // unexpected value, what could this be?

        revert(0, 0)

    }

}

}

}

interface ERC20 {

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

}

contract TestERC20SafeTransfer {

    using ERC20SafeTransfer for ERC20;

    function ping(address _token, address _to, uint _amount) {

        require(ERC20(_token).safeTransfer(_to, _amount));

    }

}

```

```

    }

}

```

另一方面，正在编写 ERC 20 合约的开发人员需要意识到这个错误，这样他们就可以预料到 BadToken 的意外行为并处理它们，这可以通过预期 BadER 20 接口并在调用后检查返回数据来确定我们调用的是 Godtoken 还是 BadToken 来实现：

```

pragma solidity ^0.4.24;

/*
 * WARNING: Proof of concept. Do not use in production. No warranty.
 */

interface BadERC20 {

    function transfer(address to, uint value) external;

}

contract BadERC20Aware {

    function safeTransfer(address token, address to , uint value) public returns (bool
result) {

        BadERC20(token).transfer(to,value);

        assembly {

            switch returndatasize()

                case 0 {                                     // This is our BadToken

                    result := not(0)                         // result is true

                }

                case 32 {                                     // This is our GoodToken

```

```

        returndatacopy(0, 0, 32)

        result := mload(0)           // result == returndata of external
call

    }

    default {                       // This is not an ERC20 token

        revert(0, 0)

    }

}

require(result);                   // revert() if result is false

}

}

```

4. 事件总结

造成本次事件的主要原因还是在于波场 USDT 的 transfer 函数未使用 TIP20 规范的写法导致函数在执行时未返回对应的值，最终返回默认的值 false，从而导致在使用 safeTransfer 调用 USDT 的 transfer 时永远都只返回 false，导致用户无法提现。

所以，在波场部署有关 USDT 的合约，需要注意额外针对 USDT 合约进行适配，上线前务必做好充足的审计与测试，尽可能减少意外事件的发生。

2.6 DeFi 通证经济设计方面的安全案例分析

DeFi 安全审计的盲区，智能合约本身没有安全漏洞，在传统审计非常难发现，必须结合经济学和数学理论进行分析发现漏洞。下面的例子是 0x 协议团队的经济学家 Peter Zeitz，发现 DeFi 项目 Curve 协议存在一个严重漏洞，并为此撰写了一份报告，根据这份报告显示，尽管 Curve 和 Swerve 协议已经过了多次合约审计，但其用户仍面临着巨大的财务损失风险。Curve 是提供稳定币之间交易的自动做市商项目，Swerve

是在 Curve 的项目进行分叉引进更加公平的流动性挖矿机制的 DeFi 项目。

下面是文章的原文翻译：

在 2020 年 9 月 19 日凌晨的几个小时，我发现了一个针对 Curve 合约的漏洞，当合约的放大系数 A 更新时，攻击者可提取大量代币余额。而使用了 Curve 合约的 Swerve，其一度更新了它的 A 系数，因此用户的潜在损失是巨大的，占到了合约余额的 36.9%，假设进行一次优化后的攻击，那么大约会损失 9200 万美元。幸运的是，Swerve 更新顺利通过，没有发生意外情况。那天下午早些时候，我通知了 Curve 团队。几个小时后，他们确认了漏洞的存在，我们开始一起研究解决方案。

实际上，这种攻击在 A 向上和向下调整时都可能发生。但是，由于向下调整的潜在损失要大一个数量级，因此我们将重点讨论这类攻击。这些攻击的严重程度与 A 的变化幅度成正比。事实证明，代币余额份额的最大损失受如下等式的限制，其中 A_{old} 是初始参数值，A_{new} 是更新的参数值，而 n 是合约中代币类型的数量。

$$\text{Max \% Loss} = 1 - \sqrt[n+1]{\frac{A_{\text{new}}}{A_{\text{old}}}} \quad (1)$$

利用漏洞造成的损失，取决于参数 A 的百分比变化

例如，yCurve 合约的更新，发生在同一周的早些时候。该合约有 n=4 个代币类型，更新从 A_{old}=2000 更改为 A_{new}=1000。使用方程 1 中的公式，攻击者可利用该漏洞提取高达 12.9% 的 yCurve 合约余额（或大约 7700 万美元）。

这种攻击只可能在预定的参数 A 更新过程中进行。Curve 合约在正常操作下不易受到攻击，因此，没有必要采取紧急行动来保护用户资金。但是，在发生其它关于 A 的更改之前修补此漏洞是至关重要的（大幅向下调整 A 尤其危险）。Curve 团队正在对更新 A 的程序进行改进，这些改进应允许 Curve 合约以安全的方式继续更新参数 A。

1. 平均数和代币联合曲线

为了理解攻击，我们有必要了解下代币联合曲线。我将解释一些概念，以便读者能够形成一个概念性的理解。我对这一主题采用了一种稍有不同的方法，重点是代币联合

曲线与一组变量平均值之间的关系。

在数学中，平均数（mean）是表示一组数据集中趋势的量数。因此，如果 x_1 是 n 个数集中最小的数， x_n 是最大的数，则这个集合的平均数将呈现为介于 x_1 和 x_n 之间的中间值。两种最常见的平均数类型是算术平均数和几何平均数。

$$\bar{x}_{AM} = \frac{1}{n} (x_1 + x_2 + \cdots x_n) \quad (2)$$

平均数在代币联合曲线中起到了关键作用。AMM 合约允许用户交易任何组合的代币，这样 AMM 合约代币余额的平均值在交易发生前后保持不变。在不同的 AMM 设计中，会使用不同类型的平均数方法。对于 Uniswap，它使用的是未加权的几何平均数，对于 Balancer，它使用的是加权几何平均数，对于 mStable，它使用的则是未加权的算术平均数。

而 Curve 使用的是算术平均数和几何平均数的加权平均数，我称之为 Curve 平均数。Curve 平均数的权重由所谓的放大参数 A 决定。随着 A 向无穷大方向增加，Curve 的平均数收敛到 mStable 使用的算术平均数。相反，如果 A 设置为 0，Curve 的平均数将与 Balancer 和 Uniswap 使用的几何平均数相同。对于 A 的中间值，Curve 的代币联合曲线将位于这两个极端的中间。

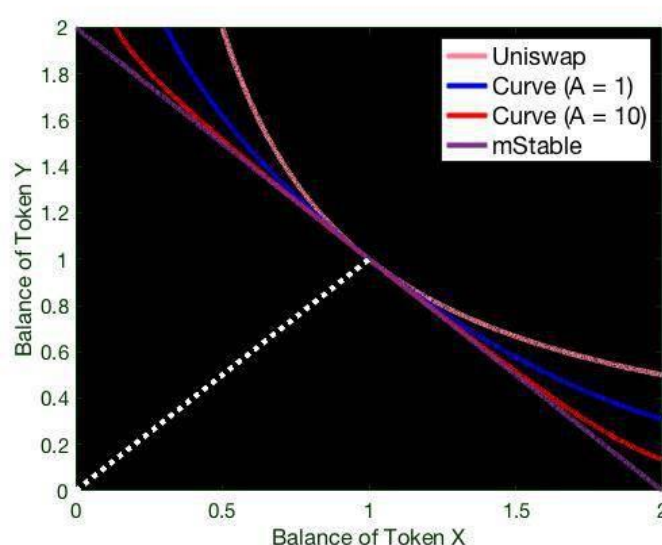


图 1 代币联合曲线

图 1 显示了四种代币联合曲线。Uniswap 保持几何平均常数，这产生了一个非常陡峭的曲率。mStable 则是算术平均值常量，它是一条直线，而 Curve 则位于两者之间。在参数值 $A=1$ 时，Curve 类似于 Uniswap，在 $A=10$ 时，Curve 更接近于 mStable。

2. 平均数和 AMM 合约持有的价值

参考图 1，我们可以看到，所有四条曲线在距离图形原点 45 度线的一个点相交。我们可以利用这个交点到原点的距离，来快速测量 AMM 合约代币投资组合的价值。例如，如果这个交叉点到原点的距离增加了 20%，那么，假设没有无常损失，AMM 合约持有的价值也将增加 20%。这适用于我们所有的四种联合曲线类型。当我们考虑 Curve 时，这一特性尤其有用，因为 Curve 具有一个独特的特性：当 A 更新时，其联合曲线的形状会发生变化。对于 Curve，我们可以使用距离原点的距离来衡量参数更新前后合约投资组合的价值。显然，如果在更新 A 之后这个距离明显减少，这将是一个严重的问题。

3. 关于参数 A 的盈亏平衡更新

再次参考图 1，假设 Curve 合约的代币余额正好位于 45 度线的交点处。当所有 Curve 代币以一比一的价格比率交易时，就会出现这种情况。从这个起点更新 A 时，就没有货币损失的风险。例如，假设 Curve 从这一点开始将参数设置 $A_{old}=10$ 改为 $A_{new}=1$ 。此更新不会更改联合曲线到原点的距离。因此，参数变化将是完全无害的，不会使 Curve 流动性提供者（LP）面临财务损失的风险。直觉上，如果初始余额不完全在交叉点，但接近于这个交点，则损失的风险仍然很小。

4. 关于参数 A 的亏损更新

现在让我们看看图 2。该图说明了当更新 A 时，攻击者如何可能操纵初始条件以实现巨大的利润。为了便于说明，我展示了从 $A_{old}=10$ 到 $A_{new}=1$ 的变化，而不是 Swerve 从 $A_{old}=1000$ 到 $A_{new}=100$ 的更新。然而，事实证明，漏洞的严重程度只取决于新旧比率，因此该数字准确地描述了 Swerve 的情况。另外，图中所示的攻击只捕获了合约代币库存的 15%。而一个完全优化的攻击将交易更极端的金额，从而可捕获多达 36.9% 的代币库存。

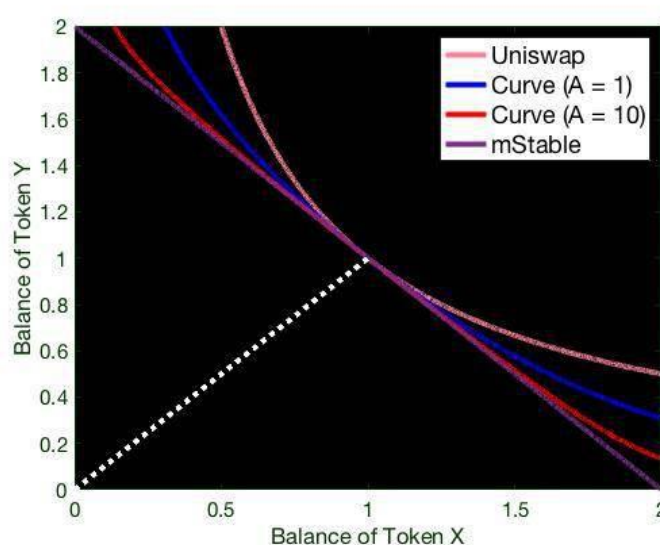


图 2 在恶意交易之间增加一个变化

假设 Curve 合约余额最初位于 45 度线的交点处，且初始参数值为 $A_{old}=10$ 。现在假设一个攻击者在两笔恶意交易之间夹了一个参数更新。在第一次恶意交易中，攻击者出售大量代币，以导致库存失衡。接下来，攻击者将触发一个更新，更新的值为 10 和 1。如图所示，这会改变曲线的形状。最后，攻击者以更低的价格买回他出售的代币。此操作将使合约沿 45 度线返回到完全平衡的状态。如图所示，此次攻击将导致 AMM 代币库存的 15% 丢失。

5. 利用漏洞的可行性

那这样的攻击真的有可能吗？令人惊讶的是，答案是 yes。Curve 合约要求提前几天安排 A 的变更，并通过去中心化的链上治理流程达成共识。但是，一旦通过治理批准了 A 中的更改，并且超过了激活截止日期，合约允许任何调用方触发更新。因此，攻击者可自由地从 Uniswap 快速租借大量稳定币，将其出售给 Curve 以触发极端不平衡，触发对 A 的更新，然后从 Curve 购买稳定币以获得巨大的利润。而完全优化的攻击会涉及到更多，这里就不再深入细节。而我上面所描述的简单攻击，就足以捕获大部分潜在利润。

6. 修复关于 A 更改的智能合约逻辑

目前，Curve 合约有两个生产版本。对于未修补的旧版合约，上面提到的内容就是漏洞的原理。而对于较新的合约，仍然存在一个潜在的漏洞，尽管其严重性要小的多。

我将首先描述旧合约的建议更改。

7. 修复旧 Curve 合约

在旧的 Curve 合约中，A 的变化发生在一个大的离散步骤中。此外，合约逻辑允许攻击者在单笔交易中以不同的 A 值执行交易。特别是，攻击者可以利用其初始交易来迫使库存极度失衡，然后触发 A 的变化，然后以更新后的 A 值执行更多交易。这使攻击者可执行涉及数以亿计资金量交易的整个攻击，而不会涉及到风险。为了解决这个问题，我建议更新旧的合约，以便只有受信任的多重签名帐户才能激活对 A 的更新。

此外，激活 A 应需要检查代币余额，以确认代币余额从广播参数更新交易的时间点起没有发生显著变化。这种余额检查可防止流氓矿工的攻击。特别是，一个流氓矿工可重新排序交易，这样他在更新 A 之前执行一笔大交易，然后在 A 更新后执行另一笔大交易。

余额检查可防止在合约处于意外不平衡状态时激活对 A 的更改，这足以保护 Curve LP 免受此类攻击。

8. 修复新的 Curve 合约

在较新的 Curve 合约中，A 的变化是在每次交易开始前以一系列离散的小步骤逐渐发生的。我的理解是每一区块只能调整一步。此外，合约要求在执行任何交易之前进行预定的步骤调整。这足以抵御普通攻击者，但不一定能抵御流氓矿工。特别是，一个流氓矿工可以连续铸造两个区块，并在两个区块中插入恶意交易。这将允许矿工在第一个区块中以较高的 A 值进行初始交易，并在第二个区块中以较低的 A 值进行最终交易。更糟糕的是，流氓矿工有一个扩展的窗口来尝试这些攻击。只要 A 还在更新过程中，流氓矿工就可以继续尝试挖取两个区块序列。

为了保护这些较新的合约，我建议将 A 中的步骤长度减小到每个区块不超过 0.1%。为什么小的步骤长度有帮助？这涉及到一个我还没有介绍的因素——Curve 合约会收取一笔费用，由于这笔费用，任何交易都会导致代币联合曲线稍微偏离原点。这也适用于攻击者的巨额交易，这使得攻击的利润略有下降。如果 A 的变化足够小，则完全优化的攻击所获得的收益，将被攻击者支付给合约的费用所抵消。因此，攻击者再也不可能通过在两笔交易之间夹杂一个变化来获利。

9. 关于安全审计和智能合约设计的经验教训

关于这种攻击，它要求设计者深入理解代币联合曲线，对于智能合约审计者来说，发现利用高度专业化知识的漏洞可能并不现实。实际上，Curve 合约已经过了多次安全审计，在我写这篇文章时，Swerve 合约刚刚通过了另一次审计。在我看来，一个通用的，可通过强力探测而不是理论检测的漏洞审计程序，将是非常有用的。为了检测这类漏洞，我建议代币联合曲线审计纳入任意两步交易程序的模拟。在这些过程中，审计人员将针对合约运行一笔随机交易，触发一个智能合约操作，然后运行另一笔随机交易。在此，智能合约操作将激活对 A 的更新。对于此漏洞，此模糊测试过程将揭示合约遭受灾难性损失的场景。然后，审计人员可以进一步调查，以了解根本原因。

对于智能合约设计师来说，了解审计的局限性是有帮助的。当合约允许一次执行一系列复杂的交互时，全面的模糊测试就变得不可行了。问题在于，用户交互的可能组合太多，我们无法探究每一种可能性。因此，限制用户在短时间内可采取操作的数量和种类是很有帮助的。这里的想法是避免创建一个非常复杂的智能合约，以至于无法通过暴力手段进行审计。

2.7 DeFi 监管安全的案例分析

监管安全案例：

2018 年据华尔街日报报道，11 月 8 日发布的新闻稿表示 EtherDelta 平台作为未注册的国家证券交易所进行操作被罚款。

根据 SEC 的命令，EtherDelta 是 ERC20 代币二级市场交易的在线平台，这是一种通常在初始代币发行（ICO）中发行的基于区块链的代币。该命令发现 EtherDelta 是未注册的国家证券交易所。EtherDelta 提供了一个市场，通过结合使用订单簿，显示订单的网站和在以太坊区块链上运行的“智能合约”，将数字资产证券的买卖双方聚集在一起。EtherDelta 的智能合约经过编码以验证订单消息，确认订单的条款和条件，执行成对的订单，并指示要更新的分布式分类帐以反映交易。在 18 个月的时间里，EtherDelta 的用户执行了超过 360 万份 ERC20 代币订单，其中包括根据联邦证券法发行的证券代币。在委员会发布其 2017 年 DAO 报告后，几乎所有通过 EtherDelta 平台下达的订单都进行了交易，该报告得出结论认为某些数字资产（例如 DAO 代币）是证券，提供这些数字资

产证券交易的平台将受到监管。SEC 要求交易所根据豁免进行注册或运营。EtherDelta 提供了各种数字资产证券的交易，但未根据豁免注册为交易所或运营。

SEC 执法部门联席主管 Stephanie Avakian 说：“EtherDelta 既具有在线国家证券交易所的用户界面，又具有底层功能，必须在美国证券交易委员会注册或有资格获得豁免。”

SEC 执法部门联席主管史蒂文·佩金（Steven Peikin）表示：“我们正在见证使用分布式账本技术在证券市场上进行重大创新的时期。”“但是为了保护投资者，这项创新使美国证券交易委员会对数字市场进行周密的监督，并强制执行现有法律。”

SEC 之前曾针对未注册的经纪交易商和未注册的 ICO 采取执法行动，包括在 EtherDelta 上交易的一些代币。

在不承认或否认调查结果的情况下，Coburn 同意了该命令，并同意支付 300,000 美元的非法所得，13,000 美元的判决前利息和 75,000 美元的罚款。委员会的命令承认了柯本的合作，委员会在决定不加重罚款时考虑了这一合作。

重要原因分析：

1) 在去中心化交易所平台交易被 EtherDelta 认为是证券的数字货币：虽然交易挂单和交易撮合，和交易本身具有点对点 and 去中心化的成分和基于以太坊作为基础链的去中心化成分，交易所与以太坊之间的网路服务器代码是中心化运营，按照 SEC 要求需要做实名认证，AML 和投资合格者验证。交易费的收入是由 EtherDelta 的运营团队中心化获得，与 UniSwap 的分配方法不同。

2) 没有在按照要求 SEC 登记证券交易所：DeFi 项目可以从这个事件吸取以下教训。

3) DeFi 平台不能交易或者存储被监管机构认为是证券的数字货币：除了交易挂单和交易撮合，和交易本身具有点对点 and 去中心化的成分和基于公链作为基础链的去中心化成分以外，交易所与以太坊之间的网路服务器代码和基础设施应该去中心化运营（交给社区）。

交易费的收入应该属于所有生态的参与者，生态参与者可以通过 DAO 投票形式决定奖励多少资金给技术团队和运营团队。

如果 DeFi 平台需要有基于证券的业务，就需要按照要求在对应的政府监管平台登记或者申请豁免。另外，我们总结合规的 DeFi 项目还需要有下面的特征：

1. 金融特征：该协议必须明确针对金融应用，例如信贷市场，通证交换，衍生工具/合成资产发行或交换，资产管理或预测市场。
2. 无需许可：该代码是开源的，允许任何一方无需经过第三方即可使用或在其之上进行构建。
3. 匿名：用户无需透露自己的身份。
4. 非托管：资产不由单个第三方托管。
5. 社区方式治理：单个实体不拥有升级决策和管理特权。如果有的话，必须存在一个可以信任的方法把个体权利移交给社区。
6. 没有预挖，没有创始人奖励，没有私募。参与者都有相同的机会。
7. 可以自证明清白，资金流向透明，系统偿付能力链上可验证。
8. 项目的所有代码，包括智能合约和客户端代码都通过第三方审计。

3. DApp 安全测试建议与工具

DApp 的呈现形式一般是 Web 应用或者移动 App，所以一般针对 Web 和移动应用的测试手段和工具都适用。凡是要交给公众用户使用的 DApp 都应该至少执行一次以下测试任务：

3.1 安全架构分析

安全架构分析应该发生在产品的设计阶段，包括分析系统每个模块的安全措施以及各个模块间的数据流程，尤其专注于当信息流跨越信任边界的时候系统如何保障隐私性和完整性。

威胁分析是安全架构分析的一个核心任务，这方面的工具包括：

微软的威胁建模工具

(<https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool>): 这个工具采用 STRIDE 威胁分类法, 让使用者可以输入产品的系统架构, 划分信任边界, 从而列出自动或手动地列出各种安全威胁, 来检验产品在设计上是否对这些威胁都有足够的防护措施。对应系统架构中所包含的微软产品的部份, 这个工具能够自动给出已知的威胁及相应的防护措施。

OWASP 的威胁建模工具 (<https://threatdragon.org/#/>): 与 OWASP 的其他项目一样, 威胁也是一个开源项目。它与用户的 Github 账号对接, 让用户可以直接在浏览器里进行威胁分析。

IriusRisk 威胁模型平台 (<https://iriusrisk.com/threat-modeling-tool/>): 这是一个功能强大的企业级威胁分析工具, 包含了大部分常用架构模块所对应的威胁清单。

SecurityCompass 的 SD Elements 工具

(<https://www.securitycompass.com/sdelements/threat-modeling/>): 该工具在很大程度上让威胁模型分析自动化, 可以让一般的开发者和架构师快速了解安全隐患, 从而在开发流程的早期就把安全防护措施植入到系统中。

3.2 静态代码扫描

静态代码扫描是使用安全工具对源代码或可执行代码进行检测, 来发现其中的安全隐患。其优点是可以涵盖全部的代码, 而缺点则是对程序的运行过程没有足够的信息, 误报率可能比较高。静态代码扫描比较适合于发现局部的安全漏洞。这方面的工具有以下几类:

交互式源代码安全检测。嵌入到程序员开发环境的交互式源代码安全检测。这种工具一般专注于程序员常犯的安全错误, 并在第一时间把错误消除掉, 包括 Synopsys 的 SecureAssist 以及 Veracode 的 Greenlight。

批处理式源代码安全扫描。这种工具会对源代码从头到尾全面扫描, 并构建代码模块之间的关系, 从而不仅发现局部的安全错误, 也能在很大程度上发现模块间互相调用时所产生的安全隐患。这方面的工具包括: HCL 的 AppScan Source (<https://www.hcltechsw.com/products/appscan/offerings/source>), MicroFocus 的 Fortify

Static Code Analyzer

(<https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview>) ,

Synopsys 的 Coverity

(<https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>),

Perforce 的 klocwork (<https://www.perforce.com/products/klocwork>)等等。

软件组成分析。由于现今的软件开发往往会调用已有的库函数和其他软件模块，这些外来的软件成分对系统的整体安全有可能带来很大的危害。软件组成分析就是查验所采用的外来软件成分有没有已知的安全漏洞，以确保软件在发布时不会含用供应链带来的安全威胁。这方面的工具包括：Checkmarx 的 CxSCA

(<https://www.checkmarx.com/products/software-composition-analysis>)，WhiteSource

(<https://www.whitesourcesoftware.com/>)，Synopsys 的 Black Duck

(<https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html>)，等等。

执行代码安全扫描。这类工具往往通过反汇编以及模式识别等技术手段来发现二进制执行代码中的安全漏洞，不仅适用于完整的应用程序，也可以用来扫描第三方提供的只有执行代码的库函数。这方面的工具包括 Veracode 的 Static Analysis

(<https://www.veracode.com/products/binary-static-analysis-sast>)，GramaTech 的

CodeSonar for Binaries (<https://www.grammatech.com/codesonar-sast-binary>) 等。

3.3 动态应用扫描

动态应用扫描是在程序的执行过程中自动地模拟恶意用户用一些可能会造成信息安全危害的输入或操作组合来实现对系统的保密性、一致性、以及可用性的颠覆。这种方法对基于 HTTP/HTTPS 协议的应用尤其有效，因为这类应用有众所周知的攻击类型，适合自动扫描。常用的动态扫描工具包括：

需要安装和单独运行的扫描软件，比如 HCL 的 AppScan Standard

(<https://www.hcltechsw.com/products/appscan/offerings/standard>)，开源项目 W3af

(<https://github.com/andresriancho/w3af>) 等。

可以单独运行也可以嵌入到 DevOps 流水线中的扫描软件：Fortify WebInspect

(<https://www.microfocus.com/en-us/products/webinspect-dynamic-analysis-dast/>)，Rapid7 的 appspider (<https://www.rapid7.com/products/appspider/>)，以及 OWASP 的 ZAP 开源工具 (<https://owasp.org/www-project-zap/>) 等等。

以软件即服务方式运行的扫描软件：Qualys Web Application Scanning (<https://www.qualys.com/apps/web-app-scanning/>)，Rapid7 的 InsightAppSec (<https://www.rapid7.com/products/insightappsec/>)，WhiteHat Security 的 Sentinel 平台 (<https://www.whitehatsec.com/platform/dynamic-application-security-testing/>) 等等。

这类工具越来越与 DevOps 紧密结合，而软件即服务的形式也越来越普遍。

3.4 手工渗透测试

自动扫描工具虽然方便，但是要针对 DApp 的具体逻辑进行安全测试，还必须用手工渗透测试的方法。手工渗透测试的工具可以让测试者修改各种输入参数，尝试不按照系统流程的顺序发出操作请求等等，来发现系统会不会出现逻辑性错误，以至造成对系统安全性的破坏。这种测试一般是要在项目开发过程中在非生产环境中进行，但在某些情况下经过授权也可以直接在生产环境中进行，但必须保证所造成的数据变更都能够得到恢复。

手工渗透测试所采用的一项主要技术是模糊测试 (Fuzzing)，就是对各项输入参数进行有规律甚至是随机的改动，来观察系统会如何回应。常用的测试工具包括：

Burp Suite (<https://portswigger.net/burp>)：这个工具包里面的几个工具可以搭配使用，从而完成应用测试的各项任务。具体的工具分别用来完成爬虫、调用代理、脆弱性扫描、自定义攻击、以及令牌随机性测试等任务。

Wfuzz (<https://github.com/xmendez/wfuzz>)：这是一个用 Python 开发的开源工具，没有 GUI 界面，只能通过命令行使用。功能方面它涵盖了大部分 Web 应用常见的安全漏洞，并支持多线程和多注入点。

3.5 生产环境中的测试与追踪

与其他的 Web 应用一样，DApp 部署到生产环境之后，项目团队需要持续检查使用

日志，如果发现信息安全方面的异常需要及时分析并更新应用。除了项目团队可以在生产环境中继续进行渗透测试之外，还可以利用漏洞汇报赏金计划（Bug Bounty）来调动社会上的资源来发现所发布 DApp 应用的安全问题。DApp 更新之后要及时通知所有的使用者尽快更新他们的版本。

另外就是要关注所使用的软件模块是否需要更新。就 DApp 具体而言，开发者很可能采用了已有的 DApp 开发框架，比如 Rimple 设计系统

（<https://rimple.consensys.design/>）和 DApparatus

（<https://github.com/austintgriffith/DApparatus>）。在 DApp 部署到生产环境之后，项目团队需要持续跟踪这些开发框架的更新，尤其是任何新发现的安全漏洞，及时修复，以免漏洞被黑客利用给用户带来损失。

3.6 针对智能合约的安全测试工具

智能合约作为 DApp 的逻辑处理后端，针对于智能合约的安全测试工作也必不可少。如下列出几个常用的开源智能合约安全测试工具：

Mythril（<https://github.com/ConsenSys/mythril>）：Mythril 是一个以太坊官方推荐的智能合约安全分析工具，使用符合执行来检测智能合约中的各种安全漏洞，在 Remix、Truffle 等 IDE 里都有集成。

Slither（<https://github.com/crytic/slither>）：Slither 是一个用 Python 3 编写的智能合约静态分析框架，提供如下功能：

- * 自动化漏洞检测。提供超 30 多项的漏洞检查模型，模型列表详见：[https://github.com/crytic/slither#detectors\[2\]](https://github.com/crytic/slither#detectors[2])。

- * 自动优化检测。Slither 可以检测编译器遗漏的代码优化项并给出优化建议。

- * 代码理解。Slither 能够绘制合约的继承拓扑图，合约方法调用关系图等，帮助开发者理解代码。

- * 辅助代码审查。用户可以通过 API 与 Slither 进行交互。

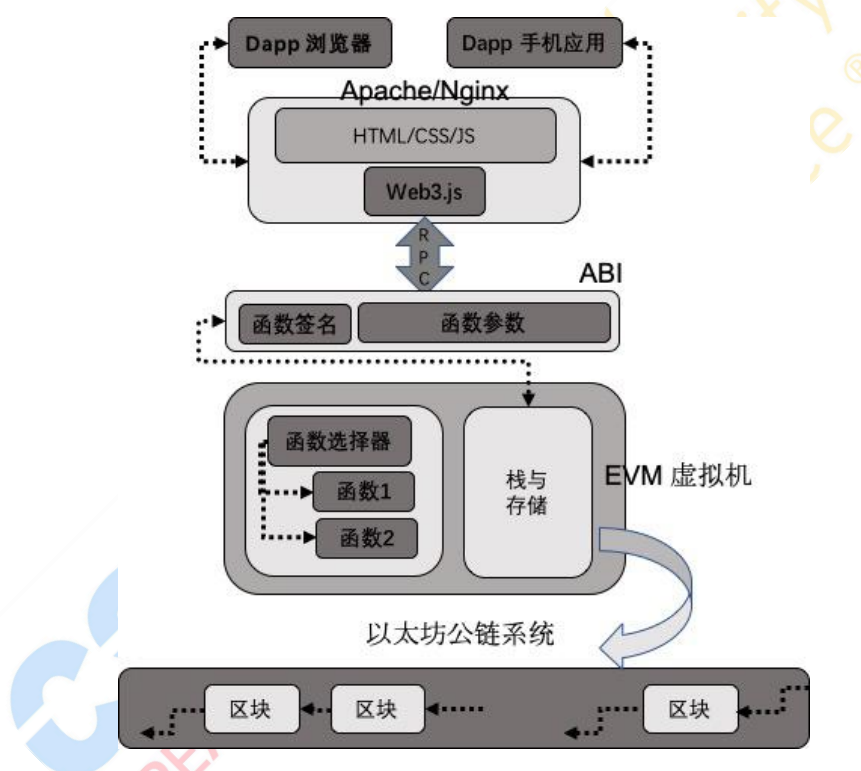
Echidna（<https://github.com/crytic/echidna>）：Echidna 是一款针对以太坊虚拟机 EVM

代码的模糊测试框架，该工具基于 Haskell 代码库实现，并支持相对复杂的基于语法的模糊测试任务。

4. DApp 安全最佳实践与案例

4.1 DApp 的基本架构

下图的基于以太坊的 DApp 的基本架构图。



首先用户通过 Web 界面或者手机 APP，将操作数据发送到一个传统的业务服务器，该业务服务器是传统互联网中心化的服务器，但是与传统系统不同的是，该系统没有像传统互联网设计那样将数据放入中心化的数据库存储，而是通过一个 Web 3.0 接口，将数据传送到了以太坊公链，该接口是一个 JSON RPC 协议，该协议有很多代码实现。目前最流行的是运行在 Web 容器中的 Web3.js 模块。Solidity 编程语言经过编译之后，除了交易需要的合约初始化代码之外，还有 ABI 接口等描述文件，Web3.js 通过这些描述文件，可以构建与以太坊智能合约虚拟机进行通讯的模块，通过 JS（全称 JavaScript，下文都简称 JS）代码将用户的操作数据传入以太坊公链上的合约地址，智能合约虚拟机

会根据函数签名和加载的函数参数，在虚拟机内执行编译成 **EVM Code** 的智能合约。如果涉及到区块链数据的读取，则虚拟机会读取区块链上的区块数据，如果虚拟机的指令代码会修改以太坊公链的状态，那么通过调用相关的状态机指令，并消耗一定的 **GAS** 之后，就可以将修改操作提交到以太坊区块链公链网络中，这些操作往往是以交易的方式体现。

当虚拟机执行结束，其对区块链状态的修改会被矿工打包，当状态修改被全网共识，那么虚拟机对公链网络的状态修改也相应成功，通过查询相关的执行结果，将执行状态返回给用户交互系统，这样终端用户就可以通过交互系统，查看 **DAPP** 操作的执行结果。

4.2 安全开发与测试的要点

DApp 开发的安全措施和最佳实践，可以总结如下：

1. 用户的私钥应该尽量容许用户控制。
2. 交易的签署应该由用户发起，服务器不能主动替用户发起交易签署。
3. 与去中心化逻辑有关的代码，比如 **DApp** 的社区投票，治理，和相关的键数据应该存储在区块链上。
4. **DApp** 的智能合约必须经过可信的第三方安全审计。
5. 在 **DApp** 上线主网之前在公共测试网上部署 **DApp**，鼓励开发者社区来测试。
6. 除了需要满足 **AML/KYC** 需求的用户数据，不能收集任何隐私方面的额外数据。
7. **DApp** 使用的智能合约，应该容许由于安全漏洞造成的升级，开发者必须把在什么情况下，需要满足什么条件，是否需要投票，升级逻辑的代码实现，等等的升级逻辑公开化。
8. **DApp** 如果需要使用外部的数据，比如通过使用预言机（**Oracle**），必须明确使用哪一个公司开发的预言机，并且对预言机可能给 **DApp** 带来的系统性风险进行详细的分析。
9. 应该在交易进行中向用户显示反馈。

4.3 DApp 开发采用 DevSecOps 流程

DApp 开发可以采用传统的 DevSecOps 流程，并确保能够执行“bug 赏金”之类的安全增强方法，从而进一步提高系统的整体安全性。有一种误解是，向公众隐藏系统的源代码可以防止缺陷和漏洞被利用。这种误解称为通过模糊实现的安全性不可取。采用 DevSecOps 的方法的重点是把安全作为第一要素在需求分析，设计，编程，测试等各个阶段进行分析和测试。在 DevOps 过程中，可以把安全测试包含在 Build Pipeline。DApp 开发可以采用传统的 DevSecOps 流程。具体课程参考美国国防部的 DveSecOps 最佳实践，链接：

https://www.dau.edu/cop/it/DAU%20Sponsored%20Documents/DevSecOps_Whitepaper_v1.0.pdf

4.4 加强安全审计

安全审计是由第三方专家或安全专家团队对智能合约代码进行的同行审查。这被证明是最有效的方法之一，因为它可以覆盖任何类型的逻辑和应用程序特定的错误。有专门的公司为智能合约安全人员提供安全审计服务。

4.4.1 开源代码评审

在开源社区发布源代码并为发现的漏洞或缺陷提供奖励。基于源代码评审可以提供奖励也是确保系统安全性的一种非常有效的方法。发现漏洞的奖励必须与审稿人执行的工作量相匹配，也必须与审稿人决定利用漏洞而不是报告漏洞时可能造成的伤害程度相匹配。

4.4.2 自动化的异常检测

在 DApp 情况下，异常检测是可以利用智能合约的属性的一个特殊功能，当智能合约中出现不需要的行为时，它会自动触发对策/警报。在某些情况下，可以预测哪种行为是恶意的，并实现“检测功能”和“合约禁用触发器”。例如，如果你开发了一个游戏程序，一个玩家连续赢了 30 次并且从来没有输过，那么很明显他可能在作弊，进一步的调查是必要的。根据类似的场景，自动化地禁用智能合约提升智能化异常检测的能

力。异常检测可以实现为一种看门狗服务，由第三方的外链服务操作，不需要在智能合约逻辑中实现。这种方法的有效性取决于智能合约开发人员确定哪种活动应该被视为“异常”的能力。这种方法对于防止特定于应用程序的错误非常好，但是它本身还不够。这种做法应该与其他安全改进措施结合使用。

4.4.3 增强手工测试

手动测试和 **testnet** 部署。测试你的软件。测试它。让智能合约开发人员和所有团队成员亲自尝试这个系统。在 **testnet** 上发布系统的一个版本，并让社区在最终版本发布前使用它几个月。这肯定会减少软件最终版本中出现漏洞的风险。这不是一种自给自足的安全改进方法，但它足够便宜和有效。



参考资料

- [1]. CSA GCR 智能合约安全指南白皮书, 2020 年 12 月出版
- [2]. 区块链安全技术指南, 机械工业出版社, 2018 年 5 月出版
- [3]. OWASP Top 10, <https://owasp.org/www-project-top-ten/>
- [4]. Harsh Agrawal, What are DApps (Decentralized Applications)? – The Beginner's Guide, <https://coinsutra.com/DApps-decentralized-applications/>
- [5]. Ethereum, <https://ethereum.org/developers/>
- [6]. Mythx, <https://mythx.io/>
- [7]. 8 个月损失 33 亿美金区块链安全问题何去何从? 除了交易所, 公链、DApp、钱包这一年也多灾多难, <http://www.lianmenhu.com/blockchain-16184-5>
- [8]. 全景扫描 2019 区块链安全事件: 数字资产被盗, 项目方跑路, <http://n.eastday.com/pnews/1577319686012065>
- [9]. BlockDelta, <https://blockdelta.io/members/audreynesbitt/>
- [10]. Gartner (Analysts Frank Catucci and Michael Isbitski) How to Deploy and Perform Application Security Testing, <https://www.gartner.com/en/documents/3982363/how-to-deploy-and-perform-application-security-testing>
- [11]. OSD DevSecOps Best Practice Guide Version 1.0, https://www.dau.edu/cop/it/DAU%20Sponsored%20Documents/DevSecOps_Whitepaper_v1.0.pdf



邮箱: info@c-csa.cn

官网: <https://c-csa.cn>