Sandboxing 相关技术研究与思考

尹至达, 高策
上海交通大学软件学院

Abstract

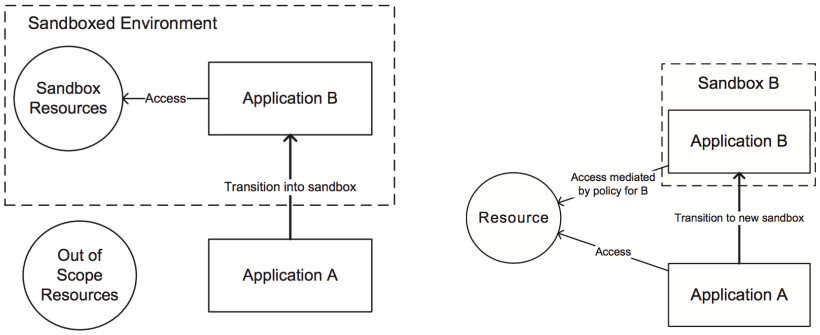
目前，互联网上的攻击越来越频繁，攻击者会通过各种攻击方式，以达到获取系统高权限，读取敏感数据等等攻击目的。沙箱作为一个用来隔离程序执行环境，使得程序在一个低权限的，高度隔离的环境中执行的技术，受到了越来越多的关注。本文以一个现代的浏览器为出发点，介绍了两类沙箱技术在其中的应用与体现，进而总结对比各种沙箱实现的特点，并且在最后对沙箱技术下一步的研究方向提出了看法。

Keywords:
Sandboxing, System Call Interposition, Capabilities, Software-based Fault Isolation

1. 引言

随着网络全球化的发展，安全越来越成为了一个进入大众视野的议题。互联网在提供给用户以便利的同时，也对隐私安全提出了新的挑战。这样的问题存在于各种平台上，攻击者会将诸如木马、病毒、恶意软件等等内容发布到被攻击者的终端上。[1] 虽然现在的操作系统上的安全措施已经足够强大到防止绝大多数的攻击，但是用户总是期望更加安全的环境。而沙箱是一种非常实用的安全机制，通过引入沙箱技术，程序可以在隔离的环境中运行，使得程序的运行不会影响操作系统，同时操作系统也不会影响到程序。在传统的操作系统的抽象中，用户的应用对于系统各种资源的使用都是无限制的，而沙箱技术是对其运行时环境的隔离 [2]。

沙箱技术大致可以被分为两类，其中第一类是基于隔离的沙箱，该类型的沙箱将应用的执行环境从操作系统中隔离出来，形成一个独立的执行环境。图 1a 展示了一个经典的基于隔离的沙箱模型。一个应用程序会在启动另一个应用程序之前先启动应用程序的沙箱，然后在沙箱内运行该应用程序，沙箱内的应用程序只能访问到沙箱内的资源。广为人知的虚拟机，容器，和传统意义上的沙箱都是这一类的沙箱模型。



(a) 第一类沙箱　　(b) 第二类沙箱

Figure 1: 沙箱分类

第二类是基于规则的沙箱，该类型的沙箱并不是完全关注于对于应用程序的隔离上，而是用规则的方式控制每个应用的权限，基于规则的沙箱之间可以分享操作系统的逻辑资源。图 1b 展示了第二类的沙箱模型，不同于基于隔离的沙箱模型的地方在于，基于规则的沙箱模型并没有实现完全的资源隔离，而是对于每个应用，有不同的限制策略，通过强制应用限制策略来保证资源的访问权限受控 [3]。

由于篇幅限制，本文不能面面俱到地对各种沙箱技术都予以介绍，为了保证深度，本文选取第一类沙箱中的 Software-based Fault Isolation，以及第二类沙箱中的 System Call Interposition 和 Capabilities，来介绍沙箱技术的应用场景与实现，以及各自的优缺点，并在文末展开对各种技术的实现原理和潜在研究点的讨论。

2. 威胁模型

目前在互联网上，不受信任的应用的数量正在快速增长，让不受信任的应用程序能够更好地运行在系统中，是一件相当困难的事情。这些不受信用的应用程序可能隐藏有带有攻击意图的代码，它们获取操作系统高权限、读取文件系统中的敏感数据（比如密码、照片、文档等）、植入广告、导致系统崩溃等等。而攻击手段有很多，其中包括代码注入、Cross Site Scripting(CSS)、缓冲区溢出、Return-oriented programming(ROP) 等等 [1]。

传统的操作系统并没有很好地解决这方面的问题，在隔离方面，仅仅依靠运行时的简单隔离是不够的。新的趋势使得操作系统应该在处理器，内存等硬件层面和进程，文件系统等软件层面进行更加细致的隔离和容错。这也是沙箱技术关注的焦点。沙箱技术有很多种，它们实现的方式也是不同的，但是其目的都是为了限制代码的运行，给予其隔离的运行环境。

// TODO：完善威胁模型

3. Sandboxing 相关技术以及使用场景

为了能够深入浅出地进行介绍，方便理解，本节将以一个现代的浏览器作为应用场景进行切入，在浏览器上，沙箱有着非常多的应用：

- 网页应用，现代的浏览器会在沙箱中运行你打开的网页代码。诸如 Javascript 之类的网页代码在浏览器的沙箱中只有有限的权限，它们并不能直接与文件系统等等进行交互，因此 Javascript 中的恶意代码的破坏性仅限于在沙箱内。
- 浏览器插件，浏览器并不会直接将插件进程运行在操作系统上，而是会将其运行在一个沙箱进程中。比如 Adobe Flash 插件，就是如此。这样的隔离方式使得插件如同网页应用一样不能直接与操作系统进行交互，防止两者之间相互的攻击。
- 各类下载内容的查看器，以 PDF 文件为例，浏览器下载的 PDF 文件会使用系统内的 PDF 阅读器打开，目前一些 PDF 阅读器正在使用沙箱技术来解析 PDF 文件，其中最有名的是 Adobe Reader。它运行在一个沙箱中，因此如果 PDF 文件中有恶意内容，也不会影响到操作系统。
- 浏览器本身，浏览器是沙箱技术运用最广泛的桌面应用之一。浏览器不止将其请求的网页应用和插件等运行在容器中，浏览器本身也是运行在一个独立而隔离的沙箱中的。低权限的沙箱运行可以保证即使恶意内容破坏了网页应用所在的沙箱，也会被浏览器的沙箱所限制[1]。

本节将介绍 System Call Interposition(SCI)、Software-based Fault Isolation(SFI) 和 Capabilities 等技术在浏览器中的体现。System Call Interposition(SCI) 通过对系统调用的跟踪和限制，来使得应用不可以调用一些敏感的系统调用，或者不能将敏感的参数传入到系统调用中。Software-based Fault Isolation(SFI) 通过对指令的重写，使得应用运行时的故障不会往外传播，同时在一定程度上限制了代码的控制流。Capability Security Model(CSM) 是一种区别于对服务提供者进行诸如 Acccess Control List(ACL) 的权限检查，而将权限检查绑定在被需要的实体上的一种设计理念[2]。这是三种完全不同的思路，它们的目的都是隔离代码的运行环境，

[1] http://www.howtogeek.com/169139/sandboxes-explained-how-theyre-already-protecting-you-and-how-to-sandbox-any-program/

[2] http://wiki.c2.com/?CapabilitySecurityModel

构建出一个高度隔离的，低特权级的沙箱环境。

3.1. System Call Interposition(SCI)

在浏览器中，有很多互联网上的内容是需要下载，然后使用本地的查看器打开的。比如 PDF 文件等，需要通过 Adobe Reader 等专用软件打开。但是下载的内容有可能是隐藏有恶意代码的，但是本地的查看器并不一定将其视为不可信的内容，因此下载的内容在被相应的查看器打开时，恶意代码可能会被错误地执行了。对 System Call Interposition（以下称为 SCI）的研究，最开始的动机就是如何限制打开浏览器下载内容的助手应用，希望能够限制助手应用本身进行系统调用的权限。Janus[4] 是 Goldberg 在 1996 年发表的论文，也是第一个成型的 SCI 系统实现，是这一领域奠基性的论文。随后有很多相关研究就此展开，研究者们继续完善 Janus，并且提出了新的解决思路。Ostia[5] 是 2004 年被提出的，它指出 Janus 存在很多缺点，而这些缺点的产生是因为架构问题，而不是 SCI 所固有的。因此它提出了一种新的架构，并且验证了他们的工作是比 Janus 的实现要高效而且可以解决 Janus 的缺点与问题。

3.1.1. Janus

// TODO

3.1.2. Ostia

// TODO

3.2. Software-based Fault Isolation(SFI)

浏览器是一个泛用性很强的应用，因此浏览器要求支持以插件的方式进行功能扩展。同时，为了提高浏览器中代码执行的性能，目前很多浏览器都在探索在浏览器中执行 Native 代码的方法。其中谷歌浏览器支持将 Native 代码运行在一个使用 Software-based Fault Isolation(以下称为 SFI) 的沙箱中。SFI 是在 1994 年在 Wahbe 的论文中第一次出现的 [6]，SFI 通过修改程序二进制的方式，使得程序的二进制代码符合一些规则，来完成对于错误的隔离。在后来的论文中，谷歌结合了 SFI 和其他的一些沙箱技术，实现了一个面向 Native 代码运行环境的沙箱，这项技术一直到现在仍在被谷歌 Chrome 浏览器使用 [7]。

3.2.1. Efficient Software-based Fault Isolation

// TODO

3.2.2. Native Client

// TODO

3.3. Capabilities

// TODO: 加一段浏览器使用场景的引入

早在 1988 年，Normhardy 就提出了 confused deputies problem[8] 问题，并借此指出了 capability 的意义与价值。Confused deputies problem 的场景可以大致描述为，一个程序需要在执行不同任务的时候，分别取得了不同的权限，这里的权限与前述的任务一一对应，但是一些恶意行为会混淆这里的权限与任务的对应，从而使得程序具有了执行不可预计的行为的权限，即其根本原因在于程序行为与其所拥有的权限之间的映射关系没能被很好的维护。但是在处理这个问题的时候，传统的基于 ACL 机制的解决方法往往会使得这个维护过程极其复杂，尤其当这里的映射关系的种类和数量变得繁多的时候，哪怕只是更新一个映射，其所带来的边际成本都会是递增的。同时，在这个问题上，"blame the user"（在这里，user 是程序本身）是不合理的，而是应该是由 OS 来应对这个安全问题[3]。Capability 作为一种被认为是可行的解决方案，其核心思想是

// TODO：核心思想 + 可靠（与 ACL 从效果上一致）

而 Capability-based Security 由于性能上的问题，大都被 OS 发行商所拒绝，很少出现在商业化的 OS 上。Capsicum[9] 作为被纳入 FreeBSD 的一个基于 capability 的安全框架，是主要的一种系统级别的实现方式，并被 Chromium 作为其在 FreeBSD 下的安全基础[4]。下文将以此作为样例，展开阐述 capability 在系统级实现中的相关细节。

3.3.1. 设计思路

// TODO: 设计思路 + 涉及到的系统改动 //记得解释 principle of least privilege 以及 ambient authority

3.3.2. 应用调整

// TODO: 应用的相应调整 + 效果

而实际上，即使不在系统级别上实现 capability-based security，依旧可以从编程的角度，进行 capability-oriented programming[5]。

4. Sandboxing 各种实现的分析与对比

5. 讨论与总结

[3] http://wiki.c2.com/?ConfusedDeputyProblem

[4] https://wiki.freebsd.org/Chromium

[5] http://wiki.c2.com/?CapabilityOrientedProgramming

参考文献

[1] S. Miwa, T. Miyachi, M. Eto, M. Yoshizumi, Y. Shinoda, Design and implementation of an isolated sandbox with mimetic internet used to analyze malwares., in: DETER, 2007.

[2] I. Borate, R. Chavan, Sandboxing in linux: From smartphone to cloud, International Journal of Computer Applications 148 (8).

[3] Z. C. Schreuders, T. McGill, C. Payne, The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls, Computers & Security 32 (2013) 219–241.

[4] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer, et al., A secure environment for untrusted helper applications: Confining the wily hacker, in: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography, Vol. 6, 1996, pp. 1–1.

[5] T. Garfinkel, B. Pfaff, M. Rosenblum, et al., Ostia: A delegating architecture for secure system call interposition., in: NDSS, 2004.

[6] R. Wahbe, S. Lucco, T. E. Anderson, S. L. Graham, Efficient software-based fault isolation, in: ACM SIGOPS Operating Systems Review, Vol. 27, ACM, 1994, pp. 203–216.

[7] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, N. Fullagar, Native client: A sandbox for portable, untrusted x86 native code, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE, 2009, pp. 79–93.

[8] N. Hardy, The confused deputy:(or why capabilities might have been invented), ACM SIGOPS Operating Systems Review 22 (4) (1988) 36–38.

[9] R. N. Watson, J. Anderson, B. Laurie, K. Kennaway, Capsicum: Practical capabilities for unix., in: USENIX Security Symposium, Vol. 46, 2010, p. 2.